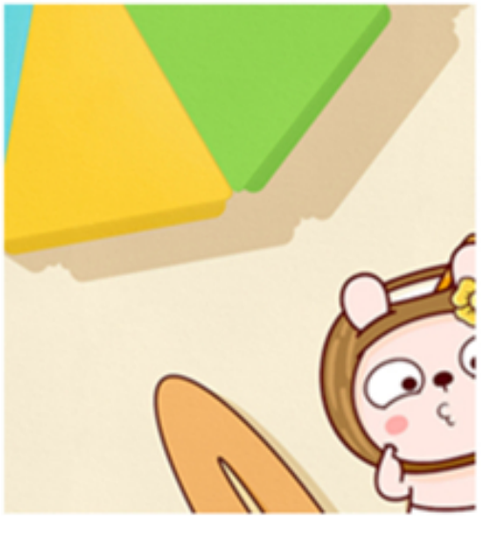
最新IT书籍PDF
最新IT书籍PDF
最新IT书籍PDF
最新IT书籍PDF



版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

非卖品，仅供非商业用途或交流学习使用

中兴通讯
技术丛书

HZ BOOKS
华章IT

OpenStack CI/CD 原理与实践

OPENSTACK CI/CD: PRINCIPLE AND PRACTICE

董文娟 尚小冬 张军◎著

由中兴通讯OPNFV开源团队撰写，团队对社区的贡献位居全球前3

从系统管理员的角度阐述OpenStack CI/CD系统的组成、架构和原理，是中兴技术团队在OpenStack CI/CD领域的实践经验总结

机械工业出版社
China Machine Press

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

内容介绍

中兴通讯OPNFV开源团队不仅技术实力雄厚，而且一直致力于为OPNFV社区做贡献，团队的贡献值在社区里排名全球前3。本书由中兴OPNFV开源团队撰写，从系统管理员的角度阐述了OpenStack CI/CD系统的组成、架构和原理，涉及从代码提交到测试、部署的各个环节，书中提到的 Gerrit 服务器管理、JJB、Zuul、Nodepool等内容在国内都鲜有介绍，希望通过本书能让你快速了解这套系统。

全书共11章，在逻辑上分为4个部分：

第一部分（第1~2章）基础篇

从宏观上对DevOps的发展、文化、工具和转型做了介绍，重点介绍了OpenStack CI/CD的概念、功能、工作流程和架构等，这是阅读本书的基础。

第二部分（第3~9章）技术篇

对OpenStack CI/CD中的每一项关键技术都进行了讲解，包括版本控制系统（Git）、代码评审系统（Gerrit）、持续集成系统（Jenkins）、门控系统（Zuul）、资源管理系统（Nodepool）、日志服务器、日志分析系统、公共组件等。

第三部分（第10章）经验篇

总结了团队在OpenStack CI/CD上积累的经验、方法和案例，以及如何进行裁剪、扩展和定制化修改。

第四部分（第11章）优化篇

总结了当前解决方案中存在的不足并给出了可行的优化方案，以及社区演进路线。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

华章IT | Information Technology

HZBOOKS

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

中兴通讯
技术丛书

OpenStack CI/CD 原理与实践

OPENSTACK CI/CD: PRINCIPLE AND PRACTICE

董文娟 尚小冬 张军◎著

机械工业出版社
China Machine Press

仅供非商业用途或交流学习使用

版权相关注意事项：

1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

非卖品，仅供非商业用途或交流学习使用

图书在版编目（CIP）数据

OpenStack CI/CD：原理与实践 / 董文娟，尚小冬，张军著．—北京：机械工业出版社，2018.10
（中兴通讯技术丛书）

ISBN 978-7-111-61191-2

I. O… II. ①董… ②尚… ③张… III. 计算机网络 IV. TP393

中国版本图书馆 CIP 数据核字（2018）第 237998 号

OpenStack CI/CD：原理与实践

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：李　艺　　责任校对：殷　虹
印　　刷：北京诚信伟业印刷有限公司　　版　　次：2019 年 1 月第 1 版第 1 次印刷
开　　本：186mm×240mm　1/16　　印　　张：18.25
书　　号：ISBN 978-7-111-61191-2　　定　　价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzit@hzbook.com

版权所有・侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所　韩光 / 邹晓东

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

Forword 推荐序一

2018 年 7 月 12 日，我正在美国波特兰出差，很荣幸收到张军的邀请，为他们团队最新撰写的书写序。与 OpenStack 相关的书籍在国内已经有很多了，而且 OpenStack 在国内的发展也进入了一个成熟期，但是特别讲 OpenStack CI/CD 或者定制化 CI/CD 系统的书籍，市场上鲜有耳闻。恰逢此时，7 月 14 日，美国商务部正式发布公告，宣布解除对中兴的拒绝令。中兴通讯也在官方微博上发布一条振奋人心的信息："满怀信心再出发！"。相信此刻中兴和上万员工的心情是激动的，我们所有人心情也是激动的。反思这件事情，我们的教训是深刻的，没有技术的自主创新和自主可控，就意味着将来可能要被别人牵着鼻子掐着脖子。因此，在我们专注的软件领域，我们要坚定不移地积极地投入开源事业。

言归正传，认识张军是通过 OPNFV（Open Platform Network Function Virtualization）社区，他是中兴通讯在 OPNFV 技术指导委员会的代表，在我的印象中，他是一个十分严谨的人。当他把本书草稿的电子版发给我后，我粗略地看了一遍。这本书是他和他的团队对

OpenStack CI/CD 系统各个组件开发和研究多年的成果，也是团队多年实践和经验的总结。除了 OpenStack 系统，现在很多软件发行版比如 Linux、Android、OPNFV，甚至是边缘计算项目 StarlingX 都可以充分利用 CI/CD 系统。正如他在书中所提到的，OpenStack CI 的框架目前只用于 OpenStack 社区基础设施的管理，了解前台的 OpenStack 的人很多，但是由于 CI/CD 是在后台的，接触和了解后面 CI/CD 的开发者却很少。所以，对于一个要全面掌握系统的开发人员来说，了解和熟悉 CI/CD 也是十分必要的。而且，CI/CD 的适用范围很广，定制化非常高，熟悉了 OpenStack CI/CD，则一通而百通。

书中除了介绍 OpenStack CI/CD 中的关键技术和组件之外，还有一大特色是分享了中兴团队在社区 CI/CD 工具集上的实践经验，同时指导读者如何定制化和修改 OpenStack CI/CD 系统，包括剪裁和扩展，即告诉读者如何以一通百。这一点是非常有益的，因为针对不同内部使用需求，针对不同项目需求，是不能直接将 OpenStack CI/CD 拿来使用的，在这点上

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

我深有体会。举个例子，我现在在英特尔公司从事边缘计算和 StarlingX 发行版的开发工作，第一个艰巨任务就是要编译和管理 StarlingX 的众多依赖包，并快速搭建起 StarlingX 的 CI/CD 体系。目前这项任务由以前负责 Linux 开发的墨西哥团队执行，同时该任务也是该边缘计算其他所有功能开发的前提和依赖。它不是简单 OpenStack CI/CD 的拷贝，在这条路上我们也费了些周折。如果墨西哥团队能了解 OpenStack CI/CD，并掌握 CI/CD 的定制化，以后在任何项目的 CI/CD 搭建道路上就可以少走很多弯路，相信本书对需要根据项目搭建并定制 CI/CD 系统的读者有很大的帮助。

另外，开源的持续集成 / 部署平台 Zuul 项目最初是为 OpenStack CI 测试开发的，后来被许多不同的组织所贡献和使用。在今年 5 月份的 OpenStack Vancouver Summit 上，OpenStack 基金会宣布 Zuul 项目发布第 3 版，并且正式成为由 OpenStack 基金会托管的第三个独立项目。这一点也足以说明 CI/CD 系统对一个项目的重要性。需要了解 Zuul 项目的读者，在本书中也能找到相应的答案。

最后，感谢张军和他的团队为 OpenStack 社区，为开源软件社区贡献了这样一本好书，希望众多 OpenStack 的开源项目的开发人员、运维人员，以及爱好者们能在书中找到它的价值，并获得帮助。

王庆博士，英特尔开源技术中心网络和存储开发经理，OpenStack 基金会个人独立董事

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

Forword 推荐序二

读了董文娟等同事创作的这本书，我感到非常高兴，并由衷感谢创作团队的杰出贡献！

云计算、大数据、人工智能是21世纪发展的基石和助推器，它们将共同促进未来生产力的提升。电信设备在往NFV、云原生演进的过程中越来越多地使用和借鉴了IT思想和工具链。这些新技术的应用，加速了整个产业链的发展。5G就是典型的综合应用场景，云原生、微服务和HTTP接口规范，奠定了5G蓬勃发展的基础。中兴通讯在云计算领域深耕多年，致力于利用先进的技术，研发更快速度、更大容量、更高安全、更具弹性、更低成本的云计算基础设施。OpenStack已经成为开源云计算领域的事实标准，其社区拥有一套完整的、标准化的、自动化的持续集成测试平台，它不仅适用于开源社区本身，也有助于电信产品的研发。本书记录了中兴通讯在这个领域的探索、研究和实践，对于构建自动化的测试系统具有很好的借鉴意义，对于公司数字化转型起到了基础支撑作用。

中兴通讯非常认同和重视开源社区，且致力于建立开放合作的生态环境。公司加入了全球多个开源的社区，成为其中的关键伙伴，比如Apache、CNCF、OpenStack和OPNFV等。中兴通讯在NFV领域技术的发展，离不开与社区的合作。中兴通讯作为最负社会责任的高科技企业之一，我们非常愿意将我们的知识、经验和服务分享到社区，回馈到社会。

本书创作团队是公司在开源社区合作的典范，是一个优秀的自组织、自管理、自激励的开放合作的敏捷团队，他们内通外联与社区合作，共同推动NFV和开源技术的发展演进。

本书是团队在 OpenStack CI/CD 领域多年的研究和实践总结，很乐意将这本优秀的著作分享给大家！

最后再次感谢创作团队的重大贡献，并欢迎各位读者开启精彩的阅读之旅！

施嵘，中兴通讯无线研究院院长

仅供非商业用途或交流学习使用

前　言 *Preface*

本书由来

过去十年，中国电信业快速发展，语音和数据业务需求极大提升，尤其当以安卓、iPhone 为代表的智能手机推出后，数据业务的增长速度远超预期。近三年，移动数据业务每年以大约 90% 的复合增长率增长[⊖]，这对设备商交付新版本的速度提出了更高的要求。中兴通讯在 2013 年就启动了无线接入网项目移植虚拟化技术预研，希望实现软硬件解耦以缩短版本交付周期，满足运营商业务飞速发展的需求。目前，这些虚拟化研究成果已经成为 5G 的基础技术标准。

虚拟化预研项目不仅涉及电信网元本身的适配，还涉及开发云操作系统和制定操作管理规范、接口标准。而云操作系统是虚拟化的关键技术之一，综合考虑云操作系统开源社区影响力、项目热度、扩展性、可维护性、业界认可度等因素，项目团队选择 OpenStack 作为云操作系统的开源解决方案。但电信领域常见的需求，如网口绑定、高速转发、巨页（Huge Page）、CPU 绑核等功能在 OpenStack 社区尚不支持，如何把虚拟化的电信网元（Virtual Network Function）运行在 OpenStack 上，并提供高性能服务，是项目面临的巨大挑战。在此背景下，笔者所在的团队开始深入参与到 OpenStack 开源社区的开发工作中，推动社区一起解决这些关键技术问题。

计算、存储和网络是 OpenStack 虚拟层三大基础设施服务，其中存储（Cinder）项目涉及厂商的硬件集成。Cinder 项目制定了一套 API 接口规范，厂商的存储设备若需要集成，需向社区提交遵守该接口规范的驱动程序，并搭建一套 CI 系统来验证，该 CI 系统需要遵守社

区第三方 CI 规范[2]。

图 0-1 是一个简化的第三方 CI 系统架构图，厂商需要准备的有：

[1] http://www.chyxx.com/industry/201708/546504.html。

[2] https://docs.openstack.org/infra/system-config/third_party.html。

仅供非商业用途或交流学习使用

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

- 第三方 CI 系统，即本书所要讲述的 OpenStack CI 系统；该系统连接厂商的存储硬件，并与社区的 Gerrit 系统[1]连接，监控 stream-events 事件流；当社区 Gerrit 有新的变更和补丁提交时，将变更代码与厂商提供的硬件环境进行集成测试并向社区 Gerrit 反馈测试结果；
- 本地 OpenStack 资源池，为第三方 CI 系统运行提供虚拟机资源；
- 存储设备，用于对变更代码进行验证。

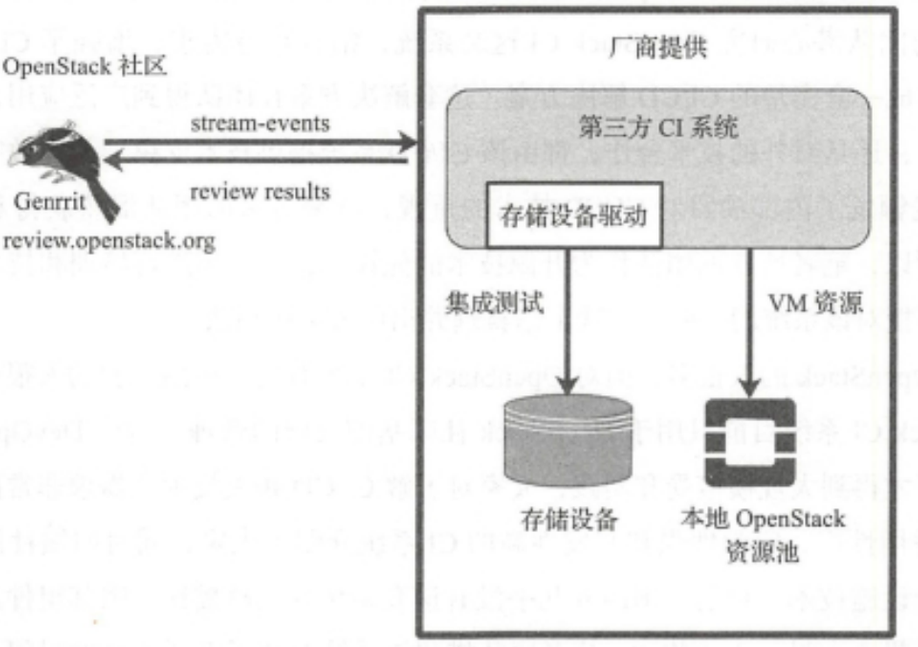


图 0-1　第三方 CI 架构图

通过这种机制，Cinder 项目可以及时发现变更代码的兼容性问题，厂商也能及时发现变更代码对设备驱动的影响。所有第三方 CI 系统的执行结果都会显示在社区 Gerrit 系统每个代码变更的评审记录中。

OpenStack CI 系统提供了一套动态调度测试任务的方法，极大提升了开发效率，具体优点如下：

- 提高 CI 系统的并发性，开发人员的多个变更测试任务不需要在环境排队等待执行，只要本地 OpenStack 资源池资源足够多，就可以并行执行所有测试任务，这可以显著减少开发人员的等待时间；
- 提高测试任务的可管理性，使用 YAML 文件描述测试任务，易阅读、评审，可进行版本管理，容易在不同团队之间复制和重用；

⊖ https://review.openstack.org。

- 可定制虚拟机基础镜像，如支持不同的操作系统、不同的软件栈配置，增加了在项目实际使用中的适用性和灵活性；
- 提高测试任务的准确性，每次测试任务都在相同的环境和配置下执行，不存在静态环境中测试任务执行后对软件、配置文件等清理不完整而导致测试任务执行不一致的问题；
- 可复用云环境，统一运维资源，减少维护开销并提高资源利用率。

这套 CI 系统不仅适用于 OpenStack 社区组件研发，也适用于电信网元的开发。基于此，笔者所在的团队潜心研究 OpenStack CI 这套系统，结合自身需求，增强了 CD 的方案和实践，最终形成一套完整的 CI/CD 解决方案。这套解决方案在团队得到广泛应用，无论是内部

的维护工作，还是对外的技术合作，都由该 CI/CD 系统提供技术支撑。公司数字化转型战略和敏捷实践触发了内部项目对 CI/CD 技术的重视，越来越多的团队渴望获得和提升这方面的技能。所以，笔者所在的团队作为开源技术的先锋，在给内部进行培训和技术转移的过程中，诞生了把对该系统的经验、实践汇总整理并出版成书的想法。

了解 OpenStack 的人很多，但对 OpenStack CI 系统有实际应用经验的人很少，主要是因为 OpenStack CI 系统目前只用于 OpenStack 社区基础设施的管理。国内 DevOps 思想和技术在最近两年才得到大规模接受和实践，大家对了解 CI/CD 相关技术的需求非常高。笔者很荣幸把这套适用性广、定制性强和并发性高的 CI 系统介绍给大家，同时回馈社区，推广和普及这些基础设施技术。目前，国内外几乎没有该套系统的完整描述，因其组件众多，且搭建该系统需要投入大量的人力物力（笔者团队搭建该系统花费了近两个月的时间，这还是在使用社区已有安装脚本进行最简安装的情况下）。从目前已经接入的第三方 CI[⊖]环境来看，国内搭建 OpenStack CI 系统的公司凤毛麟角，说明在 OpenStack 技术如此普及的情况下，国内公司对这套 OpenStack CI 系统仍然知之甚少。希望本书能让读者少走弯路，事半功倍[⊜]。

本书结构

本书从逻辑上可分为四部分：

- 第一部分（第 1 章和第 2 章）对 DevOps 的发展、文化和转型进行了简单说明，并引入本书的重点内容 OpenStack CI/CD 说明其对 DevOps 转型的重要性。
- 第二部分（第 3 ～ 9 章）以系统管理员视角对 OpenStack CI/CD 中的每一项关键技术进行详细的分析和阐述，这些关键技术包括：

⊖ https://wiki.openstack.org/wiki/ThirdPartySystems。

⊜ 本书使用 reStructuredText 语言编写，团队因采用该工具而在格式排版方面节省了大量时间精力。

  - 版本控制（Git）与代码评审（Gerrit）系统，开源社区最常用的代码管理和代码评审工具，Gerrit 系统与本书介绍的持续集成系统和门控系统进行密切配合，共同完成整个开发流程。
  - 持续集成系统（Jenkins），最为流行的一款开源持续集成工具，用于代替人工进行重复的持续集成工作，同时也为促进持续交付提供技术支撑；该章还介绍了 JJB（Jenkins Job Builder）工具，通过 YAML 文件来定义 Jenkins 任务，提升 Jenkins 任务的开发和运维效率。
  - 门控系统（Zuul），确保主线稳定，仅当与变更相关的所有测试都通过后，才会将变更合入相应的目标分支中。它保障了测试任务获得最大程度的并行，且在发生错误的情况下，自动回滚；它支持指定项目间变更依赖关系，解决跨项目集成问题，致力于守护项目及关联项目。

- 资源管理系统（Nodepool），通过在 OpenStack 资源池中动态创建虚拟机并作为 Jenkins Slave 节点添加到 Jenkins 的资源池中，确保不同的测试之间互不影响；在测试结束后，删除使用过的虚拟机，保证对每个测试任务都能提供一致的测试环境；此外，Nodepool 还集成了镜像制作功能，支持测试环境的定制开发。
- 日志服务器，对所有测试任务提供统一的日志存储服务，以便回溯测试故障。
- 日志分析系统，将测试过程中的测试日志导入 Elasticsearch 中，便于出现问题时，快速检查和分析故障。
- 公共组件，重点介绍了支持 OpenStack CI 系统组件之间进行协作和通信的系统，包括任务分发（Gearman）、消息队列（ZeroMQ）和分布式协调服务（ZooKeeper）等组件。

❑ 第三部分（第 10 章）描述团队在社区工具和经验的基础上的经典实践，以及如何进行裁剪、扩展和定制化修改以满足团队需求。

❑ 第四部分（第 11 章）探讨当前解决方案中存在的不足和可行的优化方案，并描述了社区当前正在经历的变化和未来的演进路线。

尽管本书各章节功能相对独立，笔者建议顺序阅读第 3 ～ 5 章，因为这些章节大量使用了前序章节的基本概念和术语；其他章节可以根据需要进行翻阅。

本书读者

本书面向基础设施管理团队、DevOps 团队或致力于了解 CI/CD 的技术人员，可作为学习 OpenStack 社区 CI/CD 基础设施解决方案的教材，协助你高效搭建社区第三方 CI 系统。

勘误与交流

笔者团队在编写本书的过程中虽已经进行了大量的内部评审和检验工作，但因时间紧、精力有限，难免会出现一些错漏，敬请广大专家和读者批评、指正。你可以把与本书相关的意见或建议发送到电子邮箱 openzero@aliyun.com 中。

本书中团队实践和示例代码都可以在 GitHub[⊖]上进行下载，读者可以尝试使用并与我们进行探讨。

致谢

本书的创作团队是一支拥有丰富专业技术经验的团队，团队成员有十年以上电信领域的工作经验、四年以上开源社区技术研究和贡献经验。本书是团队智慧和多年经验的结晶。使用本书介绍的技术和经验，希望可以帮助你的团队高效管理公司内部的基础设施，节省大量人力资源，并提供快速而稳定的服务。

本书由创作团队在业余时间编写，占用了团队成员很多家庭生活的时间，每位成员都付出了极大的努力并克服了诸多困难，对此表示感谢。感谢所有团队成员不厌其烦，一次又一次地评审修订，以认真严谨的态度验证了本书所有的代码实践。感谢董文娟在本书编写过程中有力地组织、协调和推进，没有你辛勤的付出，本书不可能顺利完成。

在此，还要感谢我的领导潘峰、王前和施嵘，没有你们的信任和支持，不可能有本书的诞生。感谢中兴学院的闫林院长对本书的指导和建议。感谢 OpenStack 基础设施团队设计出如此优秀的工具和系统，感谢你们在日常工作中及时答疑解惑。感谢机械工业出版社的杨福川和李艺编辑的信任和支持，及时更正了本书中较多的排版问题。

张　军

⊖ https://github.com/openzero

Contents 目　录

推荐序一

推荐序二

前言

第 1 章　DevOps ····· 1

1.1　DevOps 简介 ····· 1

1.1.1　软件开发模型 ····· 2

1.1.2　DevOps 发展历史 ····· 2

1.1.3　DevOps 循环 ····· 3

1.1.4　DevOps 价值 ····· 4

1.2　DevOps 与团队文化 ····· 4

1.3　DevOps 工具链 ····· 6

1.4　DevOps 转型 ····· 7

1.5　本章小结 ····· 8

第 2 章　CI/CD ····· 9

2.1　CI/CD 介绍 ····· 9

2.1.1　持续集成 ····· 9

2.1.2　持续交付 ····· 11

2.1.3　持续部署 ····· 12

2.1.4　CI/CD 工作流 ····· 12

2.2　OpenStack CI/CD ····· 14

2.2.1　当前 CI/CD 系统的形态 ····· 14

2.2.2　OpenStack CI/CD 架构 ····· 15

2.2.3　CI/CD 系统工作流程 ····· 18

2.3　本章小结 ····· 19

第 3 章　版本控制（Git）与代码评审（Gerrit） ····· 20

3.1　版本控制系统（Git） ····· 20

3.1.1　Git 仓库（repository） ····· 21

3.1.2　Git 分支（branch） ····· 21

3.1.3　Git 提交（commit） ····· 21

3.1.4　Git 标签（tag） ····· 22

3.1.5　Git 引用（refs） ····· 22

3.2　代码评审工具（Gerrit） ····· 23

3.2.1　架构 ····· 24

3.2.2　安装 ····· 25

3.2.3　项目配置 ····· 27

3.2.4　CI/CD 系统对接 ····· 32

3.3　本章小结 ····· 33

第 4 章　持续集成系统（Jenkins） ····· 34

4.1　Jenkins 介绍 ····· 34

4.1.1 Jenkins 是什么 …… 34
4.1.2 Jenkins 工作原理 …… 35
4.1.3 部署 Jenkins …… 37
4.1.4 插件管理 …… 39
4.1.5 安全管理 …… 44
4.1.6 创建 Slave …… 46
4.1.7 创建 Job …… 49
4.2 Jenkins Job Builder …… 57
4.2.1 安装 JJB …… 57
4.2.2 配置 JJB …… 57
4.2.3 使用 JJB …… 58
4.2.4 JJB 语法详解 …… 58
4.3 Python Jenkins …… 70
4.3.1 安装 python-jenkins …… 70
4.3.2 使用 python-jenkins …… 70
4.4 本章小结 …… 71

第 5 章 门控系统（Zuul）…… 72

5.1 Zuul 组件介绍 …… 73
5.1.1 Zuul 工作原理 …… 75
5.1.2 Zuul Server …… 76
5.1.3 Zuul Merger …… 79
5.1.4 Zuul Cloner …… 80
5.1.5 Zuul 客户端 …… 83
5.2 pipeline …… 83
5.2.1 并行测试 …… 83
5.2.2 跨项目测试 …… 85
5.2.3 跨项目依赖 …… 87
5.3 连接器 …… 88
5.3.1 Gerrit …… 89
5.3.2 SMTP …… 89
5.4 触发器 …… 90
5.4.1 Gerrit …… 90
5.4.2 Timer …… 91
5.4.3 Zuul 内部事件 …… 92
5.5 报告器 …… 92
5.5.1 Gerrit …… 92
5.5.2 SMTP …… 92
5.6 配置指导 …… 93
5.6.1 pipeline …… 95
5.6.2 Jobs …… 101
5.6.3 Projects …… 103
5.6.4 Project Templates …… 104
5.7 本章小结 …… 106

第 6 章 资源管理系统（Nodepool）… 107

6.1 Nodepool 简介 …… 107
6.1.1 Nodepool 引入的背景 …… 107
6.1.2 Nodepool 的功能 …… 108
6.2 安装 Nodepool …… 110
6.2.1 准备外部依赖服务 …… 110
6.2.2 安装 Nodepool …… 113
6.3 Nodepool 的设计原理 …… 113
6.3.1 资源管理（Nodepoold）…… 115
6.3.2 镜像管理（Nodepool-builder）…… 117
6.3.3 客户端（Nodepool Client）…… 119
6.4 配置 Nodepool …… 129
6.4.1 云相关配置 …… 129
6.4.2 Jenkins 相关配置 …… 137
6.4.3 镜像配置（diskimages）…… 140
6.4.4 其他配置 …… 142
6.5 镜像管理系统 …… 144

6.5.1 DIB 介绍 …… 145
6.5.2 DIB 原理 …… 146
6.5.3 定制镜像 …… 155
6.6 本章小结 …… 156

第 7 章 日志服务器 …… 157

7.1 日志服务器的作用 …… 157
7.2 安装和验证 …… 158
7.3 使用方法 …… 159
7.3.1 在 Jenkins 中使用日志服务器 …… 159
7.3.2 如何获取日志文件 …… 160
7.3.3 日志文件定期归档和清理 …… 161
7.4 本章小结 …… 161

第 8 章 日志分析系统 …… 162

8.1 ELK Stack 概况 …… 162
8.2 日志分析系统架构 …… 163
8.3 Log Pusher …… 165
8.3.1 处理流程 …… 165
8.3.2 配置 …… 165
8.4 Logstash Indexer …… 166
8.4.1 hello world …… 166
8.4.2 Logstash 管道 …… 167
8.4.3 管道配置 …… 167
8.4.4 管道配置实例 …… 169
8.5 Elasticsearch …… 171
8.5.1 面向文档的数据库 …… 171
8.5.2 索引、检索和搜索 …… 172
8.5.3 节点和集群 …… 178
8.5.4 索引分片和索引副本 …… 180
8.5.5 分布式特性 …… 180
8.5.6 配置 Elasticsearch …… 181
8.6 Kibana …… 183
8.6.1 让 Kibana 连接到 Elasticsearch …… 183
8.6.2 Index Pattern …… 184
8.7 部署 …… 186
8.8 本章小结 …… 186

第 9 章 公共组件详解 …… 187

9.1 任务分发系统（Gearman） …… 187
9.1.1 Gearman 介绍 …… 187
9.1.2 Gearman 架构和工作原理 …… 188
9.1.3 安装 …… 189
9.1.4 利用 Gearman 实现 Jenkins 的 HA …… 192
9.2 消息队列（ZeroMQ） …… 194
9.2.1 ZeroMQ 介绍 …… 194
9.2.2 ZeroMQ 的特点 …… 194
9.2.3 ZeroMQ 的工作模式 …… 195
9.2.4 安装 …… 196
9.2.5 应用示例 …… 196
9.2.6 ZeroMQ 在 OpenStack CI/CD 系统中的作用 …… 199
9.3 分布式协调服务（ZooKeeper） …… 199
9.3.1 ZooKeeper 介绍 …… 199
9.3.2 ZooKeeper 架构和工作原理 …… 200
9.3.3 ZooKeeper 的安装和配置 …… 203
9.3.4 ZooKeeper 典型应用 …… 205
9.3.5 Nodepool 中使用 ZooKeeper 示例 …… 208
9.4 本章小结 …… 209

第 10 章　社区 CI/CD 实践 ····· 210

10.1　Puppet 简介 ····· 210

10.1.1　概述 ····· 210

10.1.2　基础架构 ····· 214

10.2　单机部署 ····· 216

10.2.1　前期准备 ····· 216

10.2.2　安装部署 ····· 217

10.3　多节点部署 ····· 231

10.3.1　IaC ····· 232

10.3.2　配置 ····· 233

10.3.3　自动化部署 ····· 239

10.4　使用 CI/CD ····· 242

10.4.1　新增项目 ····· 242

10.4.2　提交变更 ····· 253

10.4.3　定制优化 ····· 256

10.5　CI/CD 还需要考虑的问题 ····· 271

10.6　本章小结 ····· 272

第 11 章　演进 ····· 273

11.1　存在的问题 ····· 273

11.1.1　耦合 ····· 273

11.1.2　Zuul ····· 274

11.1.3　Jenkins ····· 274

11.1.4　Nodepool ····· 274

11.2　演进 ····· 275

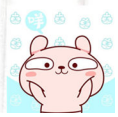
11.2.1　架构 ····· 275

11.2.2　Zuul V3 ····· 277

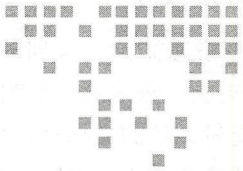
11.2.3　Nodepool V3 ····· 278

11.3　CI/CD 发展 ····· 278

11.4　本章小结 ····· 278



非卖品，仅供非商业用途或交流学习使用



第 1 章 Chapter 1

DevOps

什么是 DevOps? 为什么 DevOps 会受到越来越多的关注?

“DevOps”是英文单词“Development”和“Operation”的组合，即开发和运维的结合。目前 DevOps 并没有权威的定义，但得到大部分人认可的是，DevOps 已经成为一种文化价值观和实践方法。DevOps 价值观的呈现和实践并不依赖于特定的软件，但通过合适的软件工具链的支撑可以更容易实现这一价值观。

DevOps 使得业务可以更快地运营，更快地应对变化。2017 年 DevOps 现状调查报告[⊖]显示，工作于 DevOps 团队的人员比例达到 27%，相比于 2014 年，增加了一倍。DevOps 能够缩短上市时间，提升产品质量并增加营收，从而受到越来越多的公司关注并进行实践。

1.1 DevOps 简介

谈到 DevOps，就不得不回顾一下软件开发模型（Software Development Model）[⊜]的历史。软件开发模型是指软件开发全部的过程、活动和任务的结构框架。它包括需求、设计、编码和测试等阶段，有时也包括维护阶段。软件开发模型能清晰、直观地表达软件开发全过程，明确要完成的主要活动和任务。本章不会系统介绍曾经出现的所有软件开发模型，只选取其中较典型的瀑布模型、迭代模型、敏捷模型进行说明，以及如何演进到 DevOps。

⊖ https://ask.qcloudimg.com/raw/yehe-133f028/dorjc09gaa.pdf。

⊜ https://baike.baidu.com/item/ 软件开发模型。

非卖品，仅供非商业用途或交流学习使用

1.1.1 软件开发模型

瀑布模型严格遵循预先计划的需求分析、设计（可细分为系统设计、概要设计、详细设计）、编码、集成、测试的步骤顺序进行。上一阶段的成果输出成为下一阶段的工作输入，阶段之间通过交付大量的文档进行传递。严格分级、分阶段导致团队的灵活性降低，使项目难以响应需求的变化，难于调整或调整的代价极高。

迭代式开发模型每次只设计和实现这个产品的一部分，在迭代式开发方法中，整个开发工作被组织为一系列短小的、固定长度（如 4 周）的小周期，被称为一系列的迭代。每一次迭代中都会包含完整的需求分析、设计、编码与测试，再通过反馈来细化需求，并开始新一轮的迭代。这种方式弥补了瀑布模型中的一些弱点，加快了对变更的响应速度，可以提前识别问题，从而降低了项目风险，提升了项目成功率和生产率。

敏捷软件开发模型，是一种从 20 世纪 90 年代开始逐渐引起广泛关注的新型软件开发方法，能应对需求的快速变化。敏捷强调开发人员与业务专家之间的紧密协作，强调面对面的沟通，强调频繁交付新的软件版本，强调能够很好地适应需求变化的代码编写和团队组织方法，也更加注重软件开发中人的作用。

瀑布式开发，从需求到测试，要求每一个开发阶段都要做到最好，特别是前期阶段，设计得越完美，提交后的成本损失就越少；迭代式开发，不要求每一个阶段尽善尽美，而是先实现主要功能，再通过持续反馈逐步完善；敏捷开发，相比迭代式开发，两者都强调在较短的开发周期交付软件，但敏捷开发的周期可能更短，更强调团队的高度协作，强调适应性而非预见性。

1.1.2 DevOps 发展历史

自从人类创造了计算机，就诞生了程序员（Programmer）这个职业，从为计算机编写程序以利用其计算能力，逐渐演变为专职的实现特定功能的开发人员（Developer）。后来由于存在大量的软硬件资源需要维护，又逐渐出现专职的运维人员（Operator）这一职业。

开发人员关注更快实现功能并交付开发产品，运维人员关注如何使产品运行得更加稳定。Dev 和 Ops 的矛盾主要是面向适应性的敏捷软件交付和面向经验性的传统运维之间的矛盾。而 DevOps 打破了开发和运维两个部门、团队之间的障碍墙。

2009 年 6 月，在 Velocity 大会上，一个名为《10 + Deploys Per Day：Dev and Ops Cooperation at Flickr》[⊖]的演讲成为本次大会最大的亮点，几乎所有和 DevOps 相关的资料都引用该演讲作为 DevOps 出现的标志事件。这个演讲提出："以业务敏捷为中心，构造适应快速发布软件的工具和文化。"

第一届 DevOpsDays 于 2009 年 10 月在比利时根特举行，这次聚会开创了 DevOps 活动的先河。本次活动尽管只通过推特（Twitter）发起和召集，却出乎意料的成功。世界各

⊖ https://cdn.oreillystatic.com/en/assets/1/event/29/10 + Deploys Per Day_Dev and Ops Cooperation at Flickr Pre- sentation.pdf。

非卖品，仅供非商业用途或交流学习使用

地的开发工程师、运维工程师，还有各种 IT 管理人员、工具爱好者齐聚一堂分享相关技术和实践。从此之后，DevOps 在全球开始流行起来。近几年随着微服务和 Docker 的流行，DevOps 得到大规模的应用和实践。

敏捷和 DevOps 并不是孤立或对立的关系，其实敏捷是 DevOps 中的关键组成部分。一般而言，敏捷关注于开发过程中的实践，而 DevOps 包含从开发、持续测试、持续部署到运维的整个过程。图 1-1 可以更好地帮助我们理解它们在软件开发过程中所处的环节。

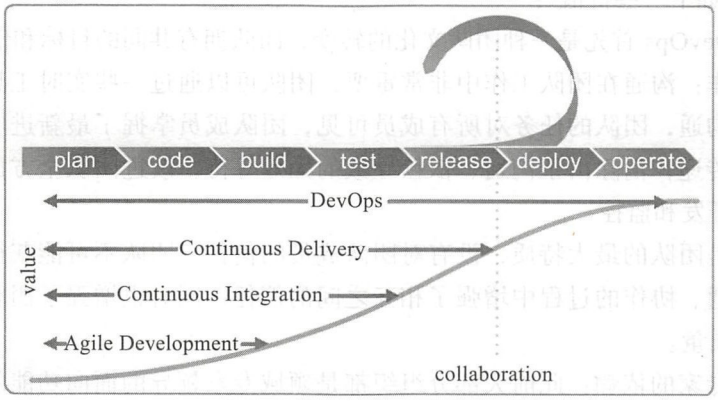


图 1-1 DevOps 生命周期

- 敏捷实践包含了从规划（涉及需求分析和设计）到代码开发和构建的过程。
- 持续集成 CI（Continuous Integration）在敏捷实践的基础上包含了持续构建和测试的过程；持续集成加强了对开发人员的反馈过程，快速验证并解决问题，提升开发效率。
- 持续交付 CD（Continuous Delivery）在持续集成的基础上，增加了持续进行版本交付的过程；持续交付加强了对测试人员的交互过程，每天都有可用的版本，避免测试阻塞，提升测试效率。
- DevOps 在持续交付的基础上，包含了持续部署和运维反馈的过程；DevOps 加强了用户体验，用户所使用的环境每天迭代更新，用户反馈的问题短时间得到更新，用户体验得到持续提升。

1.1.3 DevOps 循环

经典的 DevOps 循环如图 1-2 所示。

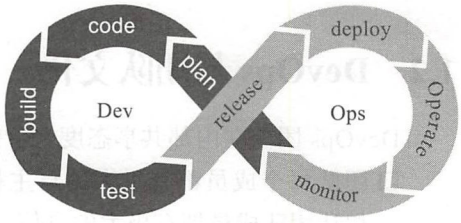


图 1-2 DevOps 循环

这个循环包含了开发和运维两部分工作，通过自动化的测试和部署把这两方面的工作进行集成。一份关于敏捷和自动化测试的报告[⊖]显示 70% 的团

⊖ https://sdtimes.com/agile/guest-view-agile-is-all-about-change-so-why-does-testing-remain-the-same。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

队采用了敏捷实践，但只有 30% 的团队采用了 CI 实践，至于 CD 就更少了。

1.1.4 DevOps 价值

DevOps 可以缩短软件发布周期，提升软件质量、安全以及反馈速度。从 2016 至 2017 年的 DevOps 现状调查变化中可以看到，2017 年 DevOps 进一步提升的目标不再是生产效率，而是质量和稳定性。

DevOps 有如下一些价值[⊖]：

- 文化：DevOps 首先是一种团队文化的转变，团队拥有共同的目标和信仰。
- 沟通协作：沟通在团队工作中非常重要，团队可以通过一些实时工具进行协作，通过持续沟通，团队的任务对所有成员可见，团队成员掌握了最新进展，可以有效进行技能传递，消除由于沟通、依赖导致的无效等待，改进团队全方面的能力，包括设计、开发和监控。
- 信任：是团队的最大特质，没有对团队成员的信任，团队不可能获得成功；成员之间在沟通、协作的过程中增强了相互之间的信任，同时也增强了团队凝聚力，提升了工作质量。
- 减少对专家的依赖：此前大部分组织都是领域专家领导的面向功能的开发团队，比如开发部、质量部、网络管理部、数据库管理部等。DevOps 通过取消这些特性团队来减少对专家的依赖；团队的每个成员都是通才，可以解决团队面临的各方面的问题。
- 系统性思考：以业务逻辑的视角来全面理解系统所实现的功能，而不只是局限于深入理解某个领域。
- 精益生产：为组织的纪律、效率和有效性设置标杆；在此过程中努力做到消除浪费、加强质量、延迟承诺、快速发布、尊重个人和全局优化。
- 自动化：所有需要重复执行的工作都努力做到自动化、代码化，经验可复制、可推广。
- 持续改进：创建一个持续改进的文化氛围，无论是困难还是收获都进行分享，努力打造学习型团队，最终得以持续改善日常工作。

1.2 DevOps 与团队文化

DevOps 团队在内部共享态度、价值、目标和实践，而这些团队通常具有如下特质：

- 团队每个成员都有极大的自主权，可以选择愿意承担的工作、任务和挑战；充分授权让团队成员拥有极大的自信，同时也能够得到快速锻炼。
- 团队成员共担责任，对于错误或失败每个人都有责任，是团队整体的错误，而不归

⊖ https://www.slideshare.net/arunmurughan/top-10-devops-values。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

责于某个成员。

- 团队成员在接受不同任务的过程中，可以更全面地了解团队承担的职责，对于团队成员实现的功能也增加了认识和理解，在此过程中，进一步锻炼了自己的能力，可以参与各种角色和功能开发。
- 团队成员分享快乐和痛苦，一同解决技术问题、跨越障碍，收获成功和喜悦，并共同分担在此过程中的痛苦和磨炼。
- 团队成员在解决困难、分担和分享的过程中学会担当，因而，每个成员都会积极处理团队需要承担的事务和需要解决的问题，进一步提升团队的主动性和自治性。

如果认为使用自动化的工具把开发和运维工作结合在一起就达到了 DevOps，或者某个特定的团队来实施 DevOps，其他团队保持传统的开发模式，其实就是打着 DevOps 旗号的伪实践，DevOps 的价值和文化很难在这种模式下表现出来并发挥作用。DevOps 是一种文化的转变，团队成员同时负责了开发、部署和运维的工作。在开发过程中不考虑如何部署，不考虑如何运维，仍然是割裂的团队和工作模式。

在向 DevOps 团队转型的过程中，可以通过表 1-1 中的指标把团队分为三个等级：

表 1-1 DevOps 团队转型过程中的分级指标

指标 \ 团队	高效能团队	中效能团队	低效能团队
部署频率	按需，一天多次	1 次 / 周～ 1 次 / 周	1 次 / 周～ 1 次 / 月
变更前置时间	小于 1 小时	1 周～ 1 月	1 周～ 1 月
故障恢复时间	小于 1 小时	小于 1 天	1 天～ 1 周
变更失败率	<15%	<15%	31-45%

变更前置时间指的是从代码提交到代码在生产环境中成功运行之间的时间；低效能团队的代码部署频率在 2016 年至 2017 年间得到了大幅提升，从 1 次 / 月 ~ 1 次 / 半年提升到 1 次 / 周 ~ 1 次 / 月。

让我们看一下高效能团队和低效能团队在表现上的差异：

- 高效能团队的部署频率是低效能团队的 46 倍。
- 高效能团队部署变更到生产环境的效率比低效能团队快 440 倍。
- 高效能团队部署故障恢复时间比低效能团队快 96 倍。
- 低效能团队部署变更失败率比高效能团队高 5 倍。
- 高效能团队人工参与的时间相当于低效能团队的 60% ～ 70%。
- 高效能团队倾向于使用主干分支，每天向主干分支合入，分支只存在若干小时，而低效能团队则采用长时间的特性分支。
- 高效能团队更倾向于使用松耦合架构，无论是团队独立开发的应用和服务，还是必须与其他团队合作开发需要交互的服务。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

中国第一份 DevOps 年度调查报告[1]中指出，国内 50% 的团队的 DevOps 实践经验都少于 1 年，同时随着团队在 DevOps 上的经验积累，自动化方面的投入增加，变更失败率可以得到逐渐降低。从 2017 年的 DevOps 现状调查报告中可以看到，在亚洲，工作于 DevOps 团队的比例大约是 10%，而在美洲则高达 54%，这也就不难解释国内经验不足的现状了。

1.3 DevOps 工具链

以下是业界在 DevOps 工具链方面的经典实践[2]，在使用时可以参考。

本节只简单列举一些主流的 DevOps 工具（如图 1-3 所示），在本书的后续章节，会对部分工具进行详细阐述。

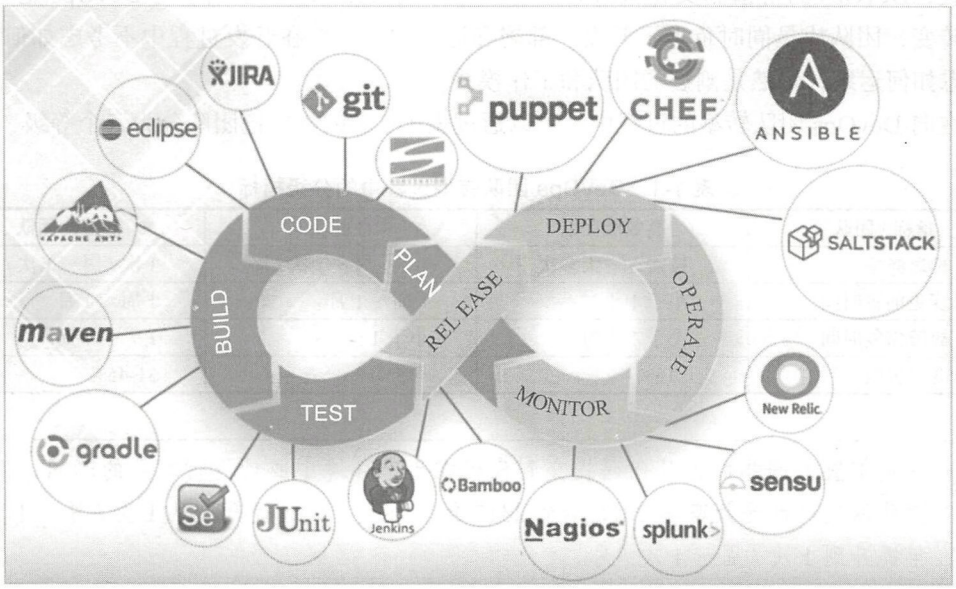


图 1-3 DevOps 工具链

（1）Git

Git 是一个自由且开源的分布式版本控制管理系统，功能丰富、性能出众，但又十分轻量，可扩展性非常好。轻量“廉价”的分支管理使其非常适合于敏捷开发，能及时响应需求和变化，开发人员可以在多任务之间轻松切换。

（2）JIRA

JIRA 是项目与事务跟踪工具，被广泛应用于缺陷跟踪、客户服务、需求收集、流程审批、任务跟踪、项目跟踪和敏捷管理等工作领域。JIRA 提供了一系列敏捷工具，如 Scrum 和

[1] http://www.10tiao.com/html/46/201712/2651000197/1.html。

[2] https://www.quora.com/What-are-the-best-tools-to-learn-DevOps-How-to-start-DevOps。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

Kanban。

（3）Maven

Maven 是一个项目管理和自动构建工具。它强调构建过程中的两个重要方面：如何构建和描述依赖关系。在实际使用中还可以通过一系列的插件来扩展各个阶段的流程。由于 Maven 包含许多重用性很强的默认规则，往往几行代码就可以完成一个项目的构建。Maven 通常用来管理 Java 项目，但也可以用于管理 C#、Ruby、Scala 等其他项目。

（4）JUnit

JUnit 是一个 Java 语言的单元测试框架。多数 Java 开发环境都已经集成了 JUnit 作为单元测试的工具。JUnit 是一套框架，使用比较简单，只要继承 TestCase 类，就可以用 JUnit 进行自动测试了。

（5）Jenkins

Jenkins 是基于 Java 开发的一款开源的持续集成工具，用于代替人工进行重复的持续集成工作，同时也为促进持续交付提供技术支撑。它支持各种版本控制工具，如 CVS、Git、ClearCase 等，并且可以与多种构建工具进行集成，如 Ant、Maven，甚至有自定义的 Shell 脚本。Jenkins 是开发和部署连接的纽带，社区活跃，有大量的插件来扩展其功能，是实现自动化的关键。

（6）Puppet

Puppet 是一款集中配置管理工具，使用自定义的 DSL 描述语言，可管理配置文件、cron 任务、软件包、系统服务等。Puppet 采用 C/S 结构，每个客户端周期性（默认半个小时）地向服务器发送请求，获得其最新的配置信息，保证系统和该配置信息同步。

（7）Ansible

Ansible 是一款新出现的自动化运维工具，基于 Python 开发，有批量配置、部署、运行命令、编排复杂任务等功能。与其他自动化运维工具的显著不同在于它只需使用 SSH 服务（SSH 可以说是各种 Linux 服务器的基本服务），它不需要在被管理节点上安装组件或代理。Ansible 本身提供了上千种 Module 可供使用。

（8）Nagios

Nagios 是一款开源的网络监视工具，能有效监控 Windows、Linux 和 UNIX 等各种主机、交换机、路由器等网络设备。在系统或服务状态异常时可以发出邮件或短信报警，第一时间通知运维人员。

1.4 DevOps 转型

DevOps 涉及文化、技术、工具等一系列支撑，业界在这方面有太多的选择，但却没有明显的解决路径，而国内这方面的人才又比较匮乏，在任何一方面的技能缺乏或没有得到合适指导的情况下，目前正在 DevOps 转型过程中的团队很容易失败。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

有些服务提供商，已经开始咨询如何进行 DevOps 转型，但痛苦的是，能胜任这项工作、有经验的顾问太少。许多公司希望寻求一个适合所有人的模式或方法，这往往会造成更多的问题。举个例子，某些公司在既不考虑这个转变是否创造了足够的业务价值，也忽略了必要的、充分的文化转变的情况下，就去构建一个 DevOps 平台。

通信、有线、娱乐和媒体公司为了跟得上市场需求，就需要有能力持续发布新产品和服务。与此同时，随着互联网普及、开源模式得到更广泛的接受，新技术的推广速度越来越快。但在短时间内实现新业务、上线新活动仍然不是一件轻松的事情，这需要强大的 IT 基础设施的支持。在 IT 所面临的挑战中[⊖]，可以看到，识别出哪些 IT 领域需要进行 DevOps 转型是非常困难的。如图 1-4 所示，IT 领域所面临的挑战大约有十项，例如：掌握开源模型和生态系统、合作伙伴管理、不充分的集成策略、缺乏清晰的技术架构路线图等，其中有 16% 受访者第一选择都是识别出哪些 IT 领域需要进行 DevOps 转型。

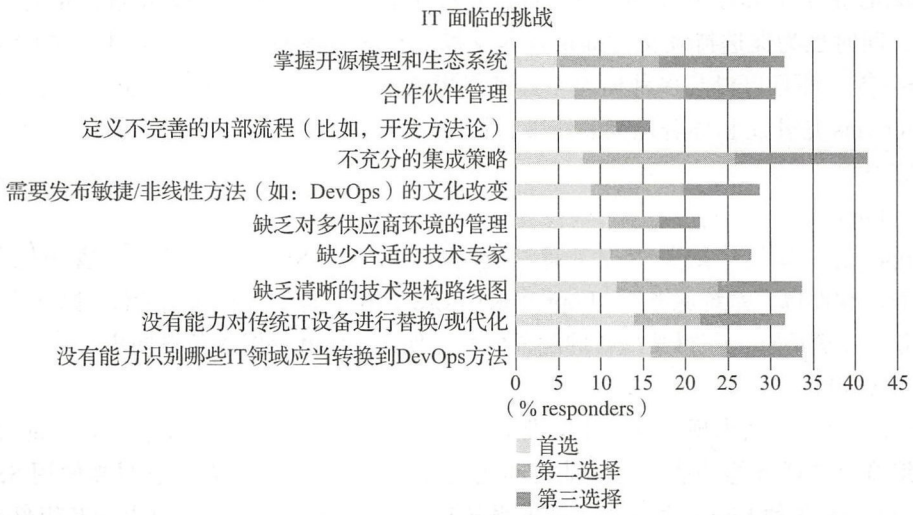


图 1-4　IT 所面临的挑战

本书将协助你了解支撑 DevOps 的技术全貌，了解当前你的团队所处的状态，并一步一步掌握转型至 DevOps 团队所需要的关键技术和技能。通过本书的学习和本书作者所在团队亲历的最佳实践的总结，希望能够协助你，选择出适合的技术路线，构建一种可扩展的、高可靠、高并发的 CI/CD 支撑平台，降低技术债包袱，逐步完成 DevOps 转型。

1.5　本章小结

本章简要介绍了软件开发模型以及 DevOps 实践的价值和转型工作，本书的后续章节将详细说明如何构建一个支撑 DevOps 转型的 CI/CD 平台。

⊖ https://www.amdocs.com/sites/default/files/2018-01/amdocs-value-of-devops-wp.pdf。

仅供非商业用途或交流学习使用

第 2 章 Chapter 2

CI/CD

DevOps 贯穿软件生命周期的全过程，而 CI/CD 则是它的基础和技术核心，在没有自动化测试、持续集成和持续部署的支撑下，DevOps 就是空中楼阁。

2.1 CI/CD 介绍

CI 是指持续集成（Continuous Integration，CI），狭义的 CD 指持续交付（Continuous Delivery，CD），广义的 CD 指持续部署（Continuous Deployment）。所谓的持续，就是说每完成一个完整的部分，就向下个环节交付，发现问题可以马上调整，使问题不会放大到其他部分和后面的环节。集成是指个人研发的软件部分向整体部分交付，以便尽早发现个人开发部分的问题；交付是指研发尽快向质量团队或者用户交付，以便尽早发现生产环境中存在的问题；部署是代码通过测试后自动部署到生产环境。

2.1.1 持续集成

随着软件项目复杂度的增加，对软件组件之间的集成和协同工作提出了更多的要求。如果开发人员在后期才进行集成，发现和解决问题的代价会很大，很有可能导致项目延期甚至失败。因此要“尽早集成、经常集成”，在项目早期发现的风险和质量问题更容易得到解决，并保证了开发进度，持续集成就诞生在这样的背景下。

持续集成指的是频繁地将代码合入到主干分支，只有通过自动化测试和验证的代码才能被集成，目的是让产品在可以快速迭代的同时还能保证质量。简单来说就是，针对软件系统的每次变更，能持续且自动地进行验证：构建、测试。根据测试结果，我们可以确定新代

仅供非商业用途或交流学习使用

码和原有代码能否正确地集成在一起，并确保核心功能正常执行，不受影响。Martin Fowler 说过："持续集成并不能消除 Bug，而是让它们非常容易被发现和改正。"

持续集成的流程如图 2-1 所示。

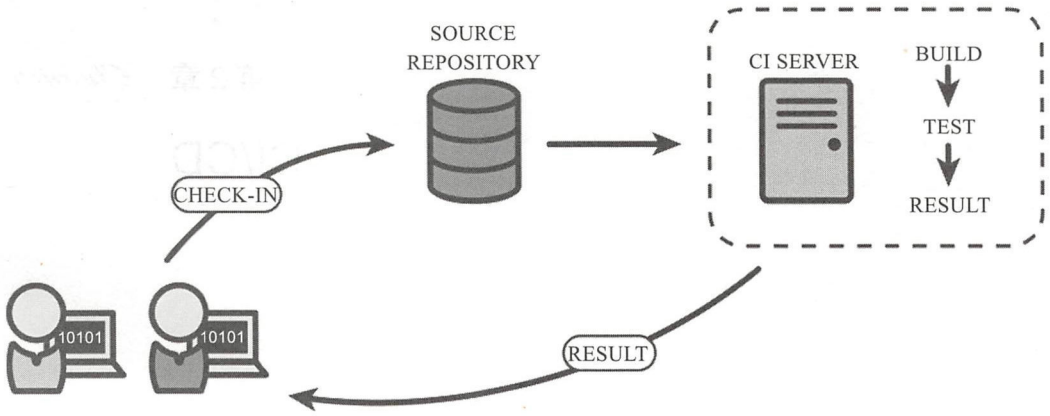


图 2-1　持续集成的流程

开发人员首先获取当前工作代码库的副本。随着其他开发人员将更改后的代码提交到源代码库，此副本逐渐与远端代码库不同步。代码库可能不仅仅被更改，还可能添加新库，或者增加其他引起依赖或者冲突的模块。

代码分支检出的时间越长，当开发者分支重新集成到主线时，集成冲突和失败的风险就越大。当开发人员将代码提交到代码库时，必须先更新代码，使本地副本和代码库中的主线分支同步。代码库包含的更改越多，开发人员在提交自己的变更前必须执行的工作越多。

开发人员的代码提交到源代码库，即触发 CI 服务器开始构建和测试，并把测试结果反馈给开发人员。测试通过，代码即合并到代码库中继续下一轮的测试；测试失败，开发人员根据反馈结果修改后重新提交。

以上过程循环重复即持续集成。

持续集成一般由代码变更触发，遵循先进先出的原则，否则容易引起混乱，很难定位是谁的代码破坏了集成；而且构建时间不能太长，否则构建次数太少，大部分变更一直处于等待状态。

需要补充一点，在代码提交之前，开发人员必须在本地完成单元测试，集成测试通常在检测到新提交时在 CI 服务器上自动运行。

持续集成的好处就在于：

- 快速发现错误，变更容易追踪，节省了项目生命周期间的时间和金钱。
- 避免版本发布最后时期才集成的混乱。
- 测试失败或出现 bug 时更容易恢复。
- 保证主干分支的可用。

❑ 频繁地集成推动开发人员设计模块化、不太复杂的代码。

可以说，持续集成是 DevOps 的重要基础环节。

2.1.2　持续交付

持续交付指在持续集成的基础上，频繁地将集成后的代码交付给质量团队（QA）或者用户，部署到更贴近真实运行环境的类生产环境中以供测试评审，通过就进入发布，生产阶段。它旨在更快更频繁地构建、测试和发布软件，有助于降低交付的成本、时间和风险。直接和可重复的部署过程对于持续交付非常重要。持续交付并不意味着每个变更都会尽快部署到生产环境中，它意味着每次更改都被证明可以随时部署。持续交付的目标不是要消灭缺陷，而是要规范开发和测试的流程，从根源上提高产品的质量。开发人员需要了解任何代码提交都可能会随时发布给客户，这一点很重要。

持续交付的流程如图 2-2 所示。

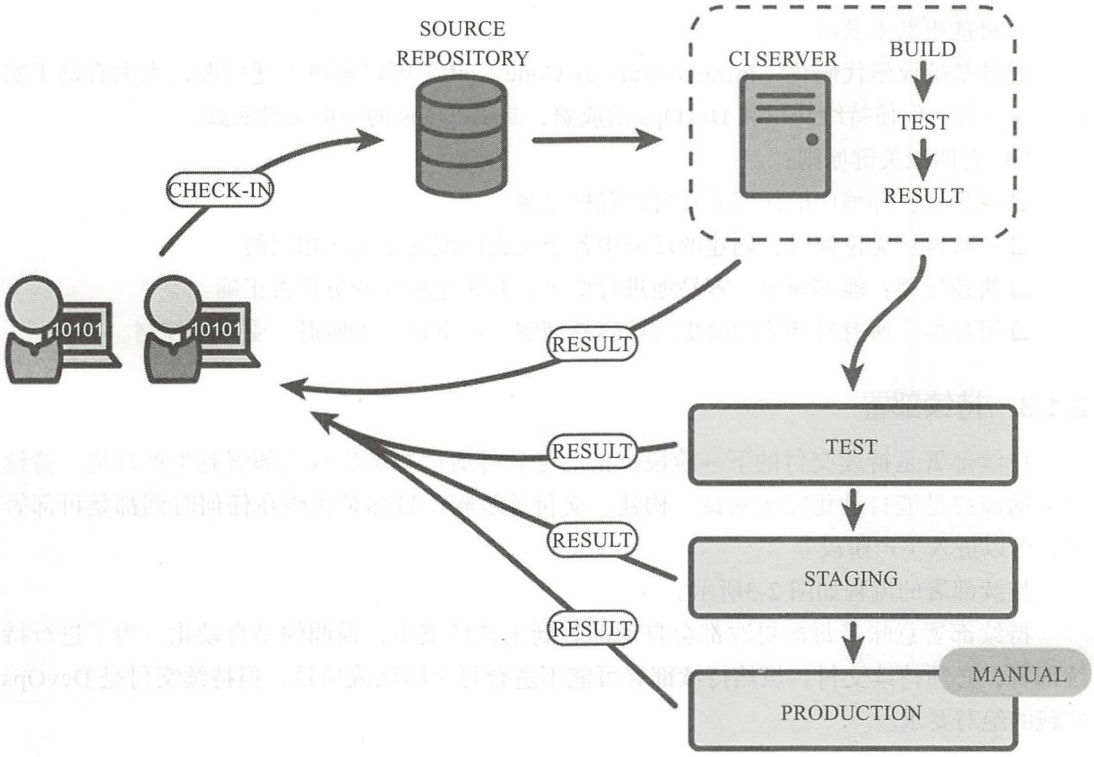


图 2-2　持续交付的流程

由图 2-2 可以看到，持续交付在持续集成的基础上多了手工部署和系统测试等流程。

持续交付的产品一般有：

❑ 源代码交付：比如 Python/Shell 脚本，但这些文件不是标准的软件包，不利于集中管

理和运行。

- 软件包交付：比如通过 RPMBuilder 工具制作 Linux 标准包。
- 镜像交付：可以是虚拟机镜像也可以是 Docker 镜像。

持续交付的优点：

- 提高了产品质量，加快了产品上市时间。
- 产品更符合用户需求，提高了客户满意度：频繁地发布使开发团队可以更快地获得用户反馈，专注于开发对客户有用的功能，有助于构建正确的产品。
- 提高了生产力和效率。

在实施持续交付的过程中，必须考虑将基础设施的维护纳入进来，作为支持产品运行的一部分。传统的基础设施运维管理存在以下几个问题：

- 自动化缺乏串联：虽然有一定的自动化，但不能做到无人值守，需要执行一些临时命令。由于环境释放和重建的成本高，因而倾向于不释放，导致资源利用率低。
- 与产品团队脱节：很难根据需求随时动态增加环境。需要额外的文档来描述环境，可能更新不及时。

通过基础设施代码化（Infrastructure as Code，IaC）可以解决上述问题，大大有助于实现持续交付，促使持续交付和 DevOps 的成熟，是 DevOps 的一项关键实践。

IaC 有四项关键原则：

- 再生性：环境中的任何元素可以轻松复制。
- 一致性：无论何时，创建的环境中各个元素的配置是完全相同的。
- 快速反馈：能够频繁、容易地进行变更，并快速验证变更是否正确。
- 可见性：所有对环境的变更应该容易理解、可审计、可回退、受版本控制。

2.1.3 持续部署

持续部署是持续交付的下一阶段，指的是软件通过测试后自动部署到生产环境。持续部署的前提是能自动化完成测试、构建、交付等步骤；目标是代码在任何时刻都是可部署的，可以进入生产阶段。

持续部署的流程如图 2-3所示。

持续部署意味着每次更改都会自动部署到生产环境中，强调的是自动化。为了进行持续部署，必须持续交付。虽然持续部署可能不适合每个团队或项目，但持续交付是 DevOps 实践的绝对要求。

2.1.4 CI/CD 工作流

以上简单介绍了 CI/CD 的概念及每个阶段的工作流程，整合起来即一套完成的 CI/CD 流水线。在实际应用中，一般通过代码评审系统实现代码审查和集成反馈。整个 CI/CD 系统配置了代码仓库系统、代码评审系统和构建工具系统。

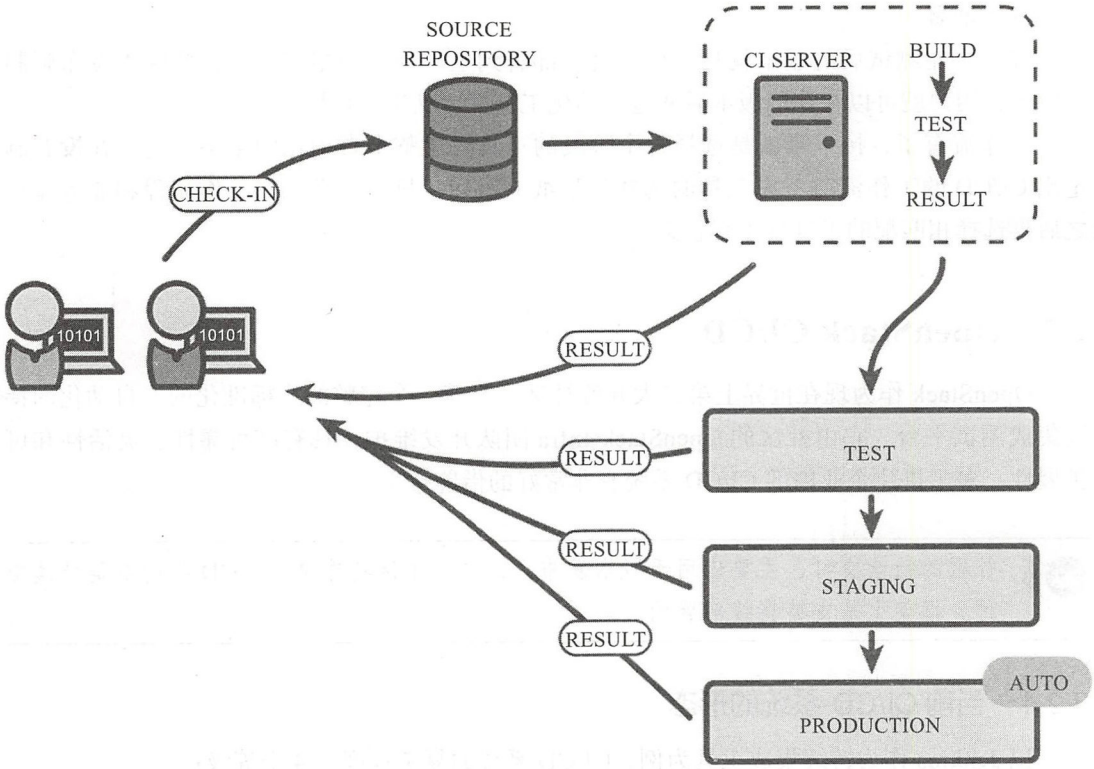


图 2-3　持续部署的流程

（1）代码提交

开发者向代码评审系统（比如 Gerrit）提交代码。

（2）测试

系统监听到代码评审系统的事件后即触发相关的测试。这里的测试有如下几种：

- 单元测试：针对函数或者模块的测试。
- 代码风格检查：针对代码编写的风格进行检查，比如 Python 的 pep8 等。
- 集成测试：功能测试。

（3）构建

测试通过后代码就可以合入主干分支，同步到代码仓库，进行下一阶段的构建了。所谓构建，就是将源代码转换为可以运行的软件包或者镜像等。

（4）测试

构建完成后进行下一轮测试，此阶段的测试比第一轮测试全面，包括功能测试、系统测试、性能测试等。

（5）交付

第二轮测试通过，代码就进入发布、生产阶段。

（6）部署

经过多轮测试后的版本就是一个可直接部署到生产环境的稳定版本，此版本发布到制品库上，用户就可以在获取版本后通过自动化工具部署到生产环境。

工作流有了，接下来就是选择每个阶段的工具搭建整个自动化的生态系统，在没有制定出 CI/CD 的工作流时考虑工具的选择只是纸上谈兵，只有在设计好工作流程和业务流程之后再选择相匹配的工具集才有意义。

2.2 OpenStack CI/CD

OpenStack 作为现在世界上第二大开源社区，有着一个完整的、标准化的、自动化的持续集成测试平台。它由社区的 OpenStack-Infra 团队开发维护，具有高可靠性、灵活性和可扩展性，对于搭建企业内部 CI/CD 系统有非常好的借鉴意义。

注解 社区的持续交付，主要应用于版本发布，后文如不做特殊说明，CD 指的都是持续交付。框架上是支持持续部署的。

2.2.1 当前 CI/CD 系统的形态

以 Jenkins 作为持续集成工具为例，CI/CD 系统的复杂程度分 4 个等级：

第 1 级：如图 2-4 所示，仅使用 Jenkins，且都在一套服务器上，适用于产品项目较小、资源较少的场景，使用维护较为简单。

第 2 级：如图 2-5 所示，在第 1 级的基础上引入了 Gerrit 评审系统，但是具体构建还是在本地，测试环境受限。

图 2-4 CI/CD 1 级

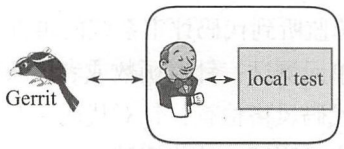


图 2-5 CI/CD 2 级

第 3 级：如图 2-6 所示，也是目前稍具规模的公司或社区都会使用的系统，使用 Jenkins + Jenkins-Job-Builder（JJB）工具。在执行构建时，使用了一些静态资源（如虚机、容器或物理机），可以满足多场景构建和测试需求，但是资源利用率较低，多次测试的环境上下文存在耦合，目前 OPNFV 社区使用这种框架。

第 4 级：如图 2-7 所示，在第 3 级的基础上引入了任务门控系统（Zuul）和动态资源管理系统（Nodepool）的使用模式，也是目前 OpenStack 开源社区的应用。

首先我们来学习一下 OpenStack 社区的 CI/CD 框架。

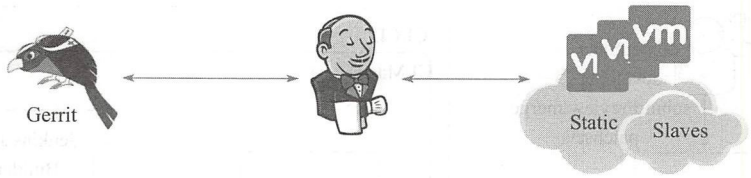


图 2-6　CI/CD 3 级

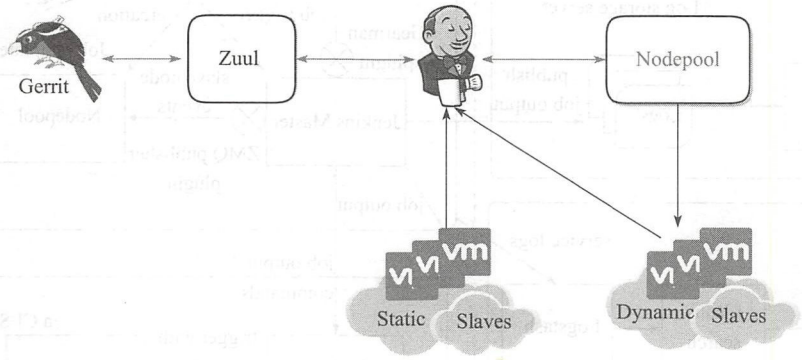


图 2-7　CI/CD 4 级

2.2.2　OpenStack CI/CD 架构

首先做个说明，社区的 CI/CD 已经演进到 V3 版本，版本之间的差异比较大，本文所有框架和组件的介绍都基于 V2 版本。

CI/CD 系统框图如图 2-8 所示。大致分为以下几部分：

- Gerrit 服务器：代码评审服务。
- CI Master：主要由以下几个组件构成。
  - 任务门控系统（Zuul）
  - 持续集成系统（Jenkins）和 Jenkins 任务管理工具（JJB）
  - 节点资源池管理系统（Nodepool）
  - 集群任务分发系统（Gearman）

这一部分是系统的核心，从监听代码变更，到匹配需要运行的任务，再到触发任务在合适的节点运行。

- CI Slave：即真正运行测试的节点，由 Nodepool 创建和管理，供 Jenkins Master 使用。
- LogServer：日志服务器。
- ELK 日志分析服务器：可视化的日志分析平台。
- 制品库：版本发布服务器。

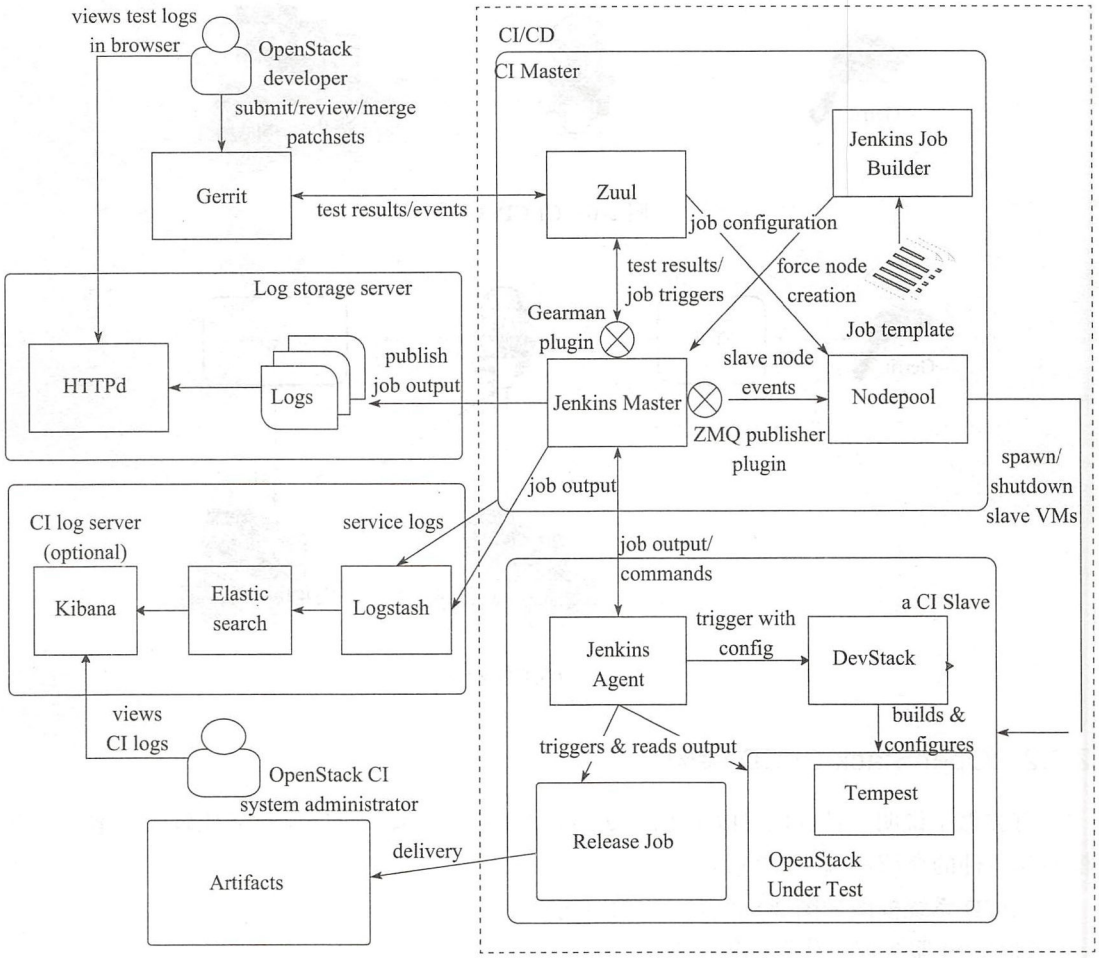


图 2-8　CI/CD 架构

下面先简单介绍各组件功能，后续章节会对每个组件做详细介绍，包括工具原理和使用方法。

（1）Gerrit

代码评审管理系统，提供了提交和补丁管理功能，同时提供了丰富的用户权限管理，控制着哪些用户或组可以提交代码、合并代码、管理代码库并提供 Gerrit 事件流通知服务。

（2）Zuul

监听 Gerrit 事件流，匹配到对应的 pipeline，以及该项目在这个 pipeline 下需要执行哪些任务。Zuul 可以处理具有复杂依赖关系的多个补丁（Patch）。它能监控正在执行的任务，并可提前结束因依赖的 Patch 测试失败而必定失败的测试任务。

（3）Jenkins & JJB

Jenkins 负责具体任务的构建，每个 Jenkins 任务都需要通过配置 Jenkins 的 config.xml

非卖品，仅供非商业用途或交流学习使用

文件实现。而在任务数量达到一定级别后，手工去配置每个任务会变得非常复杂，而且手工配置常常会带来人为操作风险。

由图 2-9 可以看出，手工干预的步数越多，部署成功率越低。

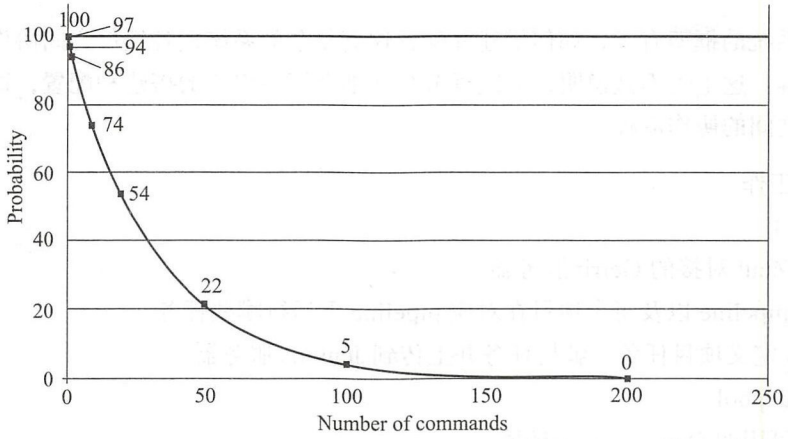


图 2-9　手工操作数 VS 部署成功率

XML 文件也不易于维护，因此引入了 JJB 这个工具。顾名思义，JJB 就是用来创建 Jenkins 任务的工具，支持 Yaml 或者 Json 格式存储 Jenkins 任务模板。JJB 通过解析用户配置的文件来自动生成 config.xml，支持模板和任务组的方式，复用性高。

（4）Gearman

Jenkins 用的是 Master/Slave 架构，一台 Master 管理所有 Slave 节点。在 Slave 节点增加到一定数量后（大约 100 台），Jenkins 的 Master 节点就会出现问题并成为瓶颈。同时 Master 节点是单点部署，无法完成 HA 等处理。为了扩展 Jenkins 而引入了 Gearman，加入 Gearman 后，Zuul 不再与 Jenkins 直接交互，而是提交执行任务的请求给 Gearman 服务器，由 Gearman 服务器完成任务的分发。通过 Gearman，使得 CI 测试架构具有了伸缩性：通过部署多个 Jenkins Master，实现了 Jenkins Master 的 HA 功能。

（5）Nodepool

管理和维护 Jenkins Slave 的服务，会在一个或多个已部署的 OpenStack 环境中自动创建 Slave 节点，并在 Slave 节点运行完一次测试后删除并重建。

（6）LogServer

日志服务器，用来存储任务构建的所有日志信息。如果测试失败，开发者可以查看日志信息排查故障，同时它也是 ELK 采集日志的源。

（7）ELK

可视化的海量日志分析平台，从日志服务器上采集日志进行存储、分析及可视化，可用于系统监控和故障排查等。

非卖品，仅供非商业用途或交流学习使用

（8）制品库

存放发布版本的服务器。

2.2.3 CI/CD 系统工作流程

CI/CD 系统的框架有了，如何搭建并配置这套复杂的系统使各组件之间协作是本书要详细阐述的内容。这里先不做说明，后续章节会详细介绍各组件的搭建和配置，这里先总体介绍下各组件之间的协作流程。

1. 准备工作

（1）Zuul

- 配置 Zuul 对接的 Gerrit 服务器。
- 定义 pipeline 以及每个项目在对应 pipeline 下运行哪些任务。

（2）JJB 定义项目任务，解析任务并上传到 Jenkins 服务器。

（3）Nodepool

- 一个可用的 OpenStack 云环境。
- 编译镜像，上传到云环境。
- 连接到云环境，用上传的镜像孵化 Slave 节点资源池。

2. 工作流

CI/CD 系统的工作流程如图 2-10 所示：

1）贡献者在 Gerrit 上提交新 Patch、添加评论等。

2）Gerrit 提交一个通知事件到它的事件流中（Event Stream）。

3）Zuul 从 Gerrit 的事件流中读取事件，并准备好本地项目代码，然后匹配事件到一个或多个 pipeline，并找到该 Patch 对应的项目下的 pipeline 任务，把任务提交给 Gearman 服务器。

4）Gearman 把 Job 分发到可以执行该任务的 Worker 上。

5）Worker 根据 Slave 节点信息在对应节点上构建任务。

6）构建后操作：把当前任务执行的日志信息和中间结果（包括编译结果）拷贝到日志服务器上。

7）返回任务结果到 Jenkins 的事件流中：

- Jenkins 通过消息方式通知 Nodepool 此次任务完成。
- Nodepool 删除执行该 Job 的 Slave 节点，并重新孵化新的节点。
- Jenkins 通过 Gearman 把当前的任务执行结果返回给 Zuul。

8）根据测试结果，Zuul 在 Gerrit 中对 Patch 添加一个 Review 结果。

9）贡献者在 Gerrit 上可以查看相关日志信息。

如果当前变更被批准合入主分支，即重复第（2）步进入持续发布阶段。

非卖品，仅供非商业用途或交流学习使用

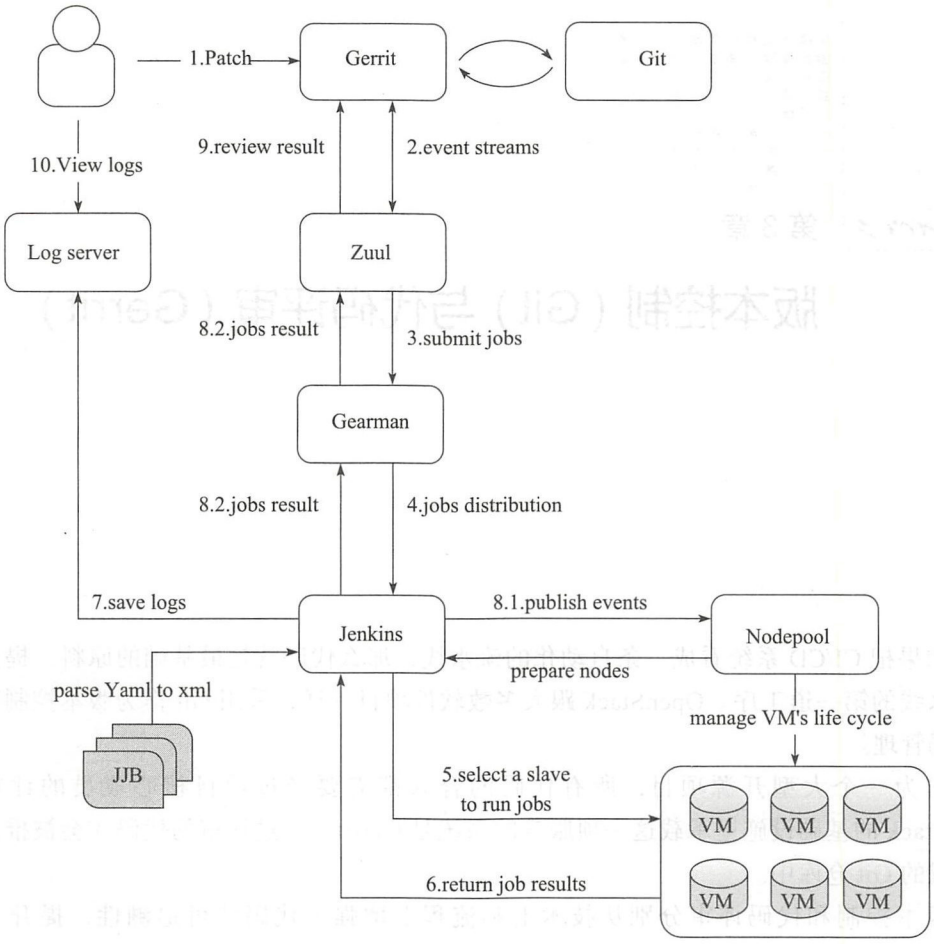
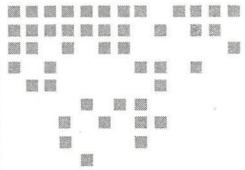


图 2-10　CI/CD 工作流程

2.3　本章小结

本章主要介绍了 CI/CD 的概念及 OpenStack 社区的 CI/CD 架构和工作流，为理解其工作原理，后续章节将详细介绍各组件的使用。

非卖品，仅供非商业用途或交流学习使用



Chapter 3 第 3 章

版本控制（Git）与代码评审（Gerrit）

如果把 CI/CD 系统看成一条自动化的流水线，那么代码就是最基础的原料，提交代码是流水线的第一道工序。OpenStack 跟大多数软件项目一样，采用 Git 作为版本控制系统进行代码管理。

作为一个大型开源项目，所有代码的合入都需要经过项目核心成员的评审。在 OpenStack 的基础设施里承载这一项服务的系统是 Gerrit。经过评审的代码才会被推送到内部集成的 Git 仓库中。

版本控制和代码评审分别从技术上和流程上增强了代码的可追溯性，提升了项目质量。

本章将介绍 Git 中的基本概念和 Gerrit 的基础配置，特别是其中与 CI/CD 相关的部分，以便于读者深入理解整个 CI/CD 系统的组成和运行机制。

3.1 版本控制系统（Git）

Git 是一个自由且开源的分布式版本控制管理系统。它的适用范围非常广泛，从简单的配置文件管理到大型的软件项目都可以使用 Git 来进行高效地管理。

尽管 Git 的功能非常丰富，但系统本身十分轻量，性能出众。凭借低成本的本地分支、灵活易用的暂存区以及多样化的工作流在各种代码管理工具如 Subversion、CVS、Perforce 以及 ClearCase 等之中脱颖而出。

本书假设读者已经对 Git 有基本的了解和使用经验，仅对与 CI/CD 相关的部分做相应的介绍。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

3.1.1　Git 仓库（repository）

Git 仓库在 CI/CD 系统里的位置如图 3-1 所示。

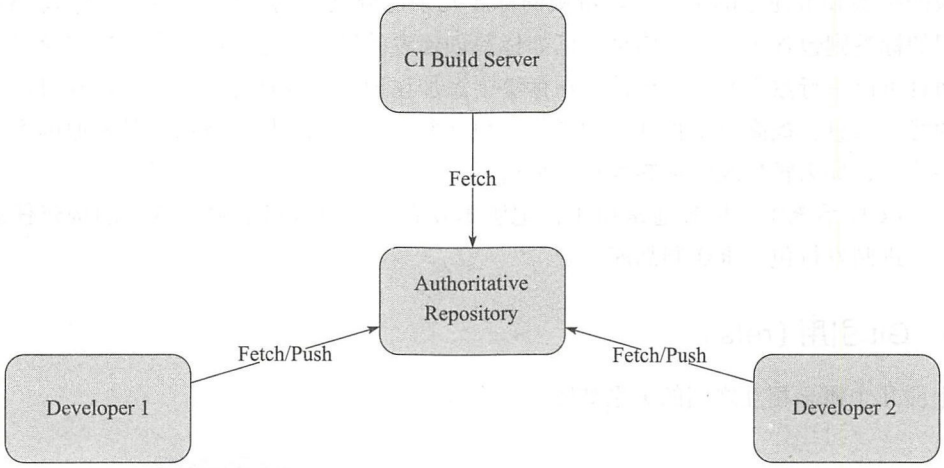


图 3-1　Git 仓库接入 CI/CD 系统

通常会部署一个中心的 Git 仓库用于存储代码，里面包含了代码的历史版本，以及各个分支所指向的版本。所有的开发者都向这个仓库提交，集成测试工具也从这个仓库获取。

3.1.2　Git 分支（branch）

几乎所有的版本控制系统都支持分支，以方便开发者独立于主线之外工作，达到一定完成程度之后再合入。有一些代码控制系统实现分支的方式是生成一份拷贝，在大型项目中这是一个相当昂贵的操作。而 Git 中的分支本质上就是对某个历史版本的引用，并不创建其他对象，创建分支几乎可以瞬时完成。

在 CI/CD 系统中，不同的分支可能会对应不同的构建任务，或者不同的环境设置。对于集成了代码评审流程的系统，未经评审的代码通常会对应一个临时分支。对于评审系统，我们会在 3.2 节详细介绍。

3.1.3　Git 提交（commit）

在 3.1.2 节我们提到，Git 分支是对某个历史版本的引用，Git 版本历史不是记录历次变更，而是一系列快照，每次代码提交都会创建当前工作区的一个快照。除此之外，Git 提交中还包含了作者姓名和电子邮件，输入的信息以及这之前一个或多个直接相邻的提交。Git 为每个提交对象生成 SHA-1 哈希，任何内容的变更都会导致 SHA-1 的变化。

在 CI/CD 系统中，通常会把被构建的分支解析为对应提交的 SHA-1 哈希。因为 SHA-1 冲突的概率低到可以忽略的程度，所以只要确定了 SHA-1，就可以追溯到唯一的构建对象。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

3.1.4 Git 标签（tag）

代码控制系统中通常使用标签来标记一些重要的历史版本。Git 支持两种类型的标签，轻量级的标签和带注记的标签。轻量级的标签几乎与分支相同，只是标签不会被轻易移动。带注记的标签则包含了更多的信息，诸如标签创建者的姓名、电子邮件和日期以及输入的信息，而且可以进行数字签名和校验。在标签中添加注记是比较推荐的做法，这样可以提供该标签的背景信息，提高可维护性。如果你只想需要一个临时的标签或由于某种原因不希望保留这些信息，那么轻量级的标签也是一种选择。

在 CI/CD 系统中，标签通常用于标记版本发布，同时也可能触发特定的构建任务，例如生成二进制发行包，推送制品库等。

3.1.5 Git 引用（refs）

上面几个概念相互之间的关系如图 3-2 所示。

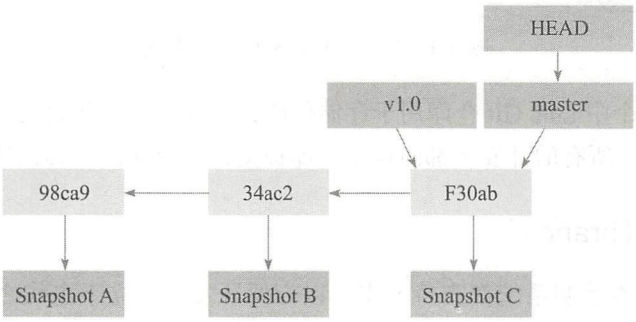


图 3-2 Git 引用

概括起来说：

- Git 仓库是一些 Git 提交（图 3-2 中的 98ca9 等）的集合。
- 每个 Git 提交都对应了代码目录的一个快照（图中的 Snapshot A 等）。
- Git 分支（图中的 master）是对 Git 提交的一个引用，并且可以被更新，以指向新的提交。
- Git 标签（图中的 v1.0）也是对 Git 提交的一个引用，它与分支非常类似，区别在于标签被设置之后通常不会再被移动。

这里我们反复提到一个概念，即“引用（reference，或缩写为 refs）”。你可以在 .git/refs 目录下找到对应的文件。如 master 对应于 .git/refs/heads/master，v1.0 对应于 .git/refs/tags/v1.0。这些文件里包含了相应提交的 SHA-1 值，即 f30ab...。Git 引用保存了 Git 提交的 SHA-1 值，提供了一个简单的名字来作为原始 SHA-1 值的替代，更方便使用。

细心的读者可能还注意到图 3-2 中的 HEAD，这个在 Git 中表示当前提交，与分支、标

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

签不同，HEAD 是一个符号引用（symbolic reference），它的内容不是原始的 SHA-1 值，而是指向其他引用。例如：

```
$ cat .git/HEAD
ref: refs/heads/master
```

CI/CD 系统就是通过上述各种不同的引用来取得特定的代码版本进行测试，有的情况下还会生成新的引用（例如创建一个临时引用，把代码和主分支合并）。更详细的内容会在第 5 章进行介绍。

3.2　代码评审工具（Gerrit）

Gerrit 是一个基于 Web 的代码评审工具，其内部采用 Git 作为版本控制系统。

代码评审在软件开发中的重要性无须赘述。Gerrit 的目的是提供一个轻量级的框架，评审每一个将被合入代码库的提交。提交到 Gerrit 中的代码只有在被评审和接受后，才真正成为项目的一部分。增加 Gerrit 之后，CI/CD 系统就变为图 3-3 所示的结构。

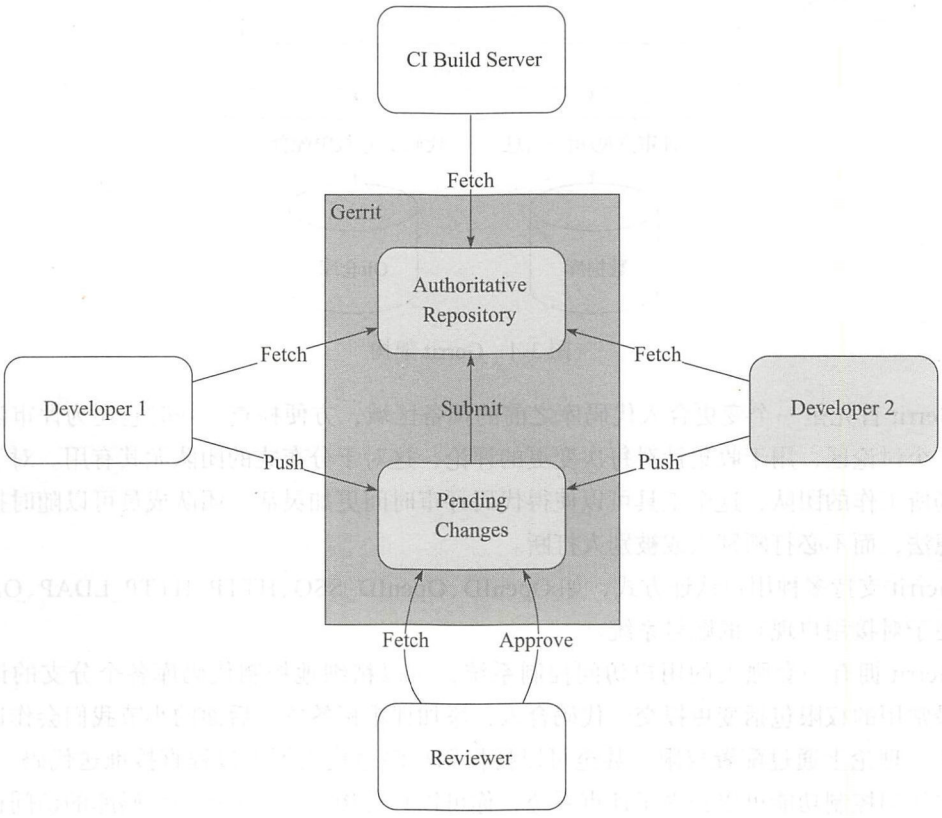


图 3-3　Gerrit 对接 CI/CD 系统

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

对比图 3-1，显著的差别在于开发者（Developer）不再直接将代码推送（Push）到代码仓库，而是需要经过评审者（Reviewer）的批准（Approve）才能真正提交（Submit）。

3.2.1 架构

了解了 Gerrit 在 CI/CD 系统中的位置和主要作用后，我们再来深入内部分析一下它的基本架构，如图 3-4 所示。

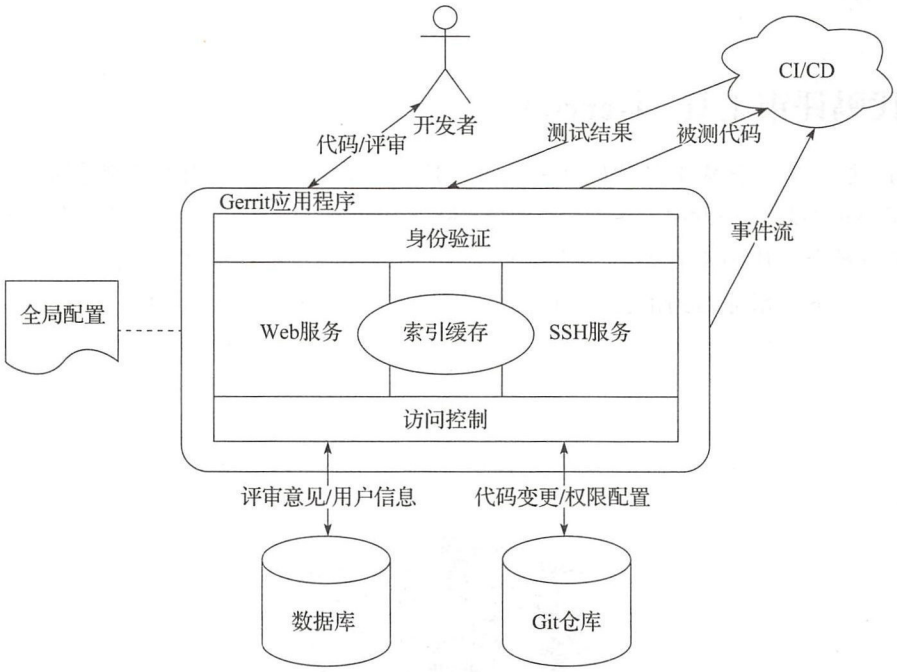


图 3-4 Gerrit 架构

Gerrit 首先是一个变更合入代码库之前的预备区域，方便检查。同时它还为评审流程提供了一个讨论区，用于收集针对每次变更的评论，这对于分布式的团队尤其有用。对于在同一个场所工作的团队，这个工具可以使得代码评审时间更加灵活。团队成员可以随时提交他们的想法，而不必打断别人或被别人打断。

Gerrit 支持多种用户认证方式，如 OpenID、OpenID_SSO、HTTP、HTTP_LDAP、OAUTH 等，便于对接用户现有的账号系统。

Gerrit 拥有一套强大的用户访问控制系统，可以精细地控制代码库各个分支的访问权限。最常用的权限包括变更提交、代码合入，添加评审标签等，后面的小节我们会作进一步的说明。理论上通过配置权限，甚至可以授权用户略过代码评审过程直接推送代码。Gerrit 的用户访问控制功能可以独立于评审系统，你可以只使用 Gerrit 来控制代码库的访问权限而不使用它的代码评审系统。通常情况下，即便是对于可以直接推送代码的用户，通过完整的

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

评审流程合入代码会更为安全。

3.2.2　安装

下面，我们从环境要求、下载方式及程序初始化等方面来介绍一下 Gerrit 的安装过程。

1. 环境要求

运行 Gerrit 需要 JDK1.7[㊀]以上的版本。除此之外，还需要一个 SQL 数据库用于存储评审中的元数据。你可以使用内置的嵌入式 H2 或者部署自己的 MySQL 或 PostgreSQL。

2. 下载 Gerrit

Gerrit 的当前版本和历史版本都可以从 Gerrit 的发布站点[㊁]下载。从中选择一个版本的 *.war 包（下文中以 gerrit.war 指代）。

如果你更偏好从代码编译，可以参考官方的开发者文档[㊂]。

3. 初始化数据库

Gerrit 会在初始化阶段要求你指定数据库类型。

（1）H2

H2 是一个嵌入式的数据库，在 Gerrit 中使用不需要额外的配置步骤。这是体验 Gerrit 的最简单方法，也适用于规模较小的团队使用。但对于规模较大的公司来说，这并不是一个推荐方案。因为 H2 很难在 Gerrit 离线的情况下进行交互操作，数据不方便备份，也无法配置负载均衡。

（2）PostgreSQL

这个选项要比 H2 复杂，推荐用于大规模安装。PostgresSQL 是 Gerrit 社区中使用最为广泛的数据库后端。

在 PostgreSQL 中为 Gerrit 创建一个新用户，指定密码。创建一个数据库 reviewdb 用于存储元数据，并授予该用户此数据库的完全访问权限。

```
createuser --username=postgres -RDIElPS gerrit2
createdb --username=postgres -E UTF-8 -O gerrit2 reviewdb
```

上述命令行中的参数含义可以参考 PostgreSQL 文档[㊃]。

（3）MySQL

这个选项也同样比 H2 复杂，与 PostgreSQL 一样同样推荐用于大规模安装。

创建一个新用户，指定密码，创建一个数据库用于存储元数据，并授予该用户完全的访问权限，如代码清单 3-1 所示。

㊀ http://www.oracle.com/technetwork/java/javase/downloads/index.html。
㊁ https://www.gerritcodereview.com/download/index.html。
㊂ https://gerrit-documentation.storage.googleapis.com/Documentation/2.14.7/dev-readme.html。
㊃ http://www.postgresql.org/docs/9.1/interactive/index.html。

仅供非商业用途或交流学习使用

代码清单 3-1 MySQL 初始化

```
mysql

CREATE USER 'gerrit2'@'localhost' IDENTIFIED BY 'secret';
CREATE DATABASE reviewdb;
GRANT ALL ON reviewdb.* TO 'gerrit2'@'localhost';
FLUSH PRIVILEGES;
```

在 shell 下运行 MySQL 客户端，然后执行相应命令即可。这里的示例适用于 MySQL 和 Gerrit 安装在同一个节点。关于 MySQL 的更多信息可以参考 MySQL 文档[㊀]。

（4）其他

Gerrit 也支持 Oracle、DB2、SAP MaxDB 等，配置方法可以参考官方文档中关于数据库创建[㊁]的部分。

4. 初始化 Gerrit 站点

Gerrit 以网站的形式对外提供服务，在使用前需要进行一些初始化工作。Gerrit 的配置文件、服务器的 SSH 密钥以及受控的 Git 仓库都保存在一个本地目录里，通常被称作 $site_path。如果使用了 H2 数据库，那么数据文件也保存在该目录下。

此外，你需要选择一个保存 Git 仓库的目录，既可以是一个 $site_path 下的相对路径，也可以是系统的绝对路径。

初始化一个新的站点目录通过运行 init 命令来完成。目标路径由 -d 选项指定。此外推荐为 Gerrit 创建一个专门的用户用于执行这些命令：

```
sudo adduser gerrit2
sudo su gerrit2

java -jar gerrit.war init -d /path/to/your/gerrit_application_directory
```

注意 如果 gerrit2 用户没有目标路径的访问权限，那么你需要手工创建目录，并把所有权设置为 gerrit2 用户。

如果使用一个交互终端运行，那么 init 命令会提示一系列的配置问题。如果所在的终端不支持交互，那么 Gerrit 会自动选择一些合理默认选项，并使用 H2 嵌入式数据库。在初始化过程结束后，你可以打开配置文件 $site_path/etc/gerrit.config 查看配置信息。

在 init 命令的运行过程中，可能需要额外下载一些 JAR 包来支持可选的功能。万一下载失败，屏幕上会打印目标 URL，等待用户手工下载后放到指定位置。

㊀ http://dev.mysql.com/doc/。

㊁ https://gerrit-documentation.storage.googleapis.com/Documentation/2.14.7/install.html#createdb。

仅供非商业用途或交流学习使用

初始化阶段结束后，守护进程会自动在后台启动，并打开浏览器访问站点：

```
Initialized /home/gerrit2/review_site
Executing /home/gerrit2/review_site/bin/gerrit.sh start
Starting Gerrit Code Review: OK
Waiting for server to start ... OK
Opening browser ...
```

打开浏览器后，通过 Web 界面登录 Gerrit。当第一个用户登录并注册时，系统会自动创建一个账号，归属于拥有全部权限的管理员组，允许其通过 Web 和 SSH 进行管理工作，后续的用户会被自动注册为非特权用户。

5. 开发模式

我们可以通过 Docker 快速搭建一个开箱即用的 Gerrit 服务器，使用了内置的 H2 数据库和开发账号：

```
docker run -ti -p 8080:8080 -p 29418:29418 gerritcodereview/gerrit
```

这个镜像可以直接用于开发和学习环境，如果想应用于生产环境，还需要定制配置文件，挂载外部存储等。

3.2.3　项目配置

Gerrit 的配置选项众多，我们在此介绍一下最常用的几个方面，包括项目、分支、标签和权限等。

1. 创建项目

新项目可以通过 Web 界面上的 Projects → Create Project 进行，如图 3-5 所示。注意，这个菜单项只对拥有该权限的用户可见。在开发环境中，我们可以登录为 admin 用户，拥有全部系统权限。

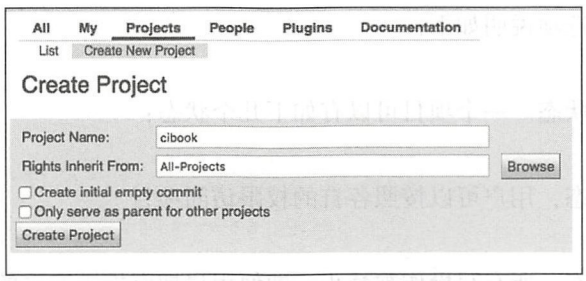


图 3-5　在 Gerrit 中创建项目

此外，也可以通过 REST 或 SSH 接口创建项目。

以 SSH 为例，在实验环境中创建一个新的名为 tools/gerrit 的项目：

```
ssh -p 29418 admin@localhost gerrit create-project tools/gerrit.git
```

仅供非商业用途或交流学习使用

注解 你必须先将 SSH 公钥添加到 Gerrit 对应用户的账号（如 admin）才能正常登录。

完整的参数说明可以参考 REST 接口创建项目[⊖]和 SSH 命令创建项目[⊜]相关文档。如果想了解 Gerrit 的内部工作机制，也可以尝试手工创建项目。

在 Gerrit 的基础路径（配置项 gerrit.basePath）下建立一个 Git 仓库：

```
git --git-dir=$base_path/tools/project.git init
```

如果需要开启匿名 git:// 协议访问，则需要建立一个空文件 git-daemon-export-ok：

```
touch   $base_path/tools/project.git/git-daemon-export-ok
```

然后在 Gerrit 中注册该项目，可以通过重启服务，或者强制刷新 project_list 缓存：

```
ssh -p 29418 admin@localhost gerrit flush-caches --cache project_list
```

2. 项目选项

项目配置可以通过网页进行，如图 3-6 所示。

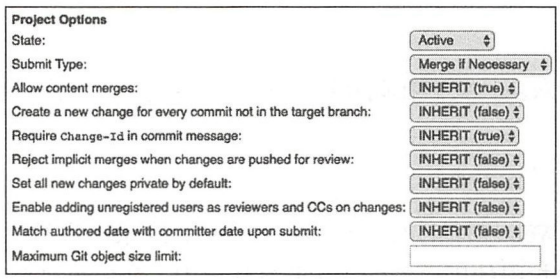


图 3-6 Gerrit 项目配置选项

其中主要的配置选项说明如下：

（1）状态 State

定义项目的全局状态。一个项目可以有如下几个状态：

❑ Active

项目处于活跃状态，用户可以按照各自的权限访问项目。

❑ ReadOnly

项目处于只读状态，所有写操作被禁止，即便用户拥有推送变更的权限操作也会失败。只读状态通常用来临时关闭一个项目，这样你可以保留所有的权限设置不变。需要重新启用时，只需要将项目状态重新设置为 Active。

⊖ https://gerrit-review.googlesource.com/Documentation/rest-api-projects.html#create-project。

⊜ https://gerrit-review.googlesource.com/Documentation/cmd-create-project.html。

另外一个使用场景是项目被迁移到其他位置。你希望所有的开发活动都在新项目下进行，老项目仅作参考，此时需要防备有人不小心提交变更到老项目。

❑ Hidden

项目处于隐藏状态，只对其所有者可见。其他用户即便拥有读的权限也无法访问。

（2）提交类型 Submit Type

设置 Gerrit 可用的变更提交方式。通常情况下，只有该变更所有的依赖都被同时提交才能成功合入，但也有一些例外的情况，下文会做介绍。

❑ Inherit

这个是新建项目的默认选项，可以通过设置全局的 defaultSubmitType 重载。在这个配置下，来自于父项目的提交类型会被继承。在最顶层的 All-Projects 里，由于没有项目可被继承，这个选项等同于 Merge If Necessary。

❑ Fast Forward Only

这个方式将不会产生合并提交，所有的合并操作都必须在本地完成，然后才可以上传到 Gerrit 中评审。

变更必须是目标分支的严格超集才能被合入，也就是说，变更的祖先中必须包含目标分支的顶端，而不是从基于某个历史版本产生分叉。

❑ Merge If Necessary

如果变更是目标分支的严格超集，那么分支会被快速推进到该变更。否则的话，自动产生一个合并提交。这个与经典的 git merge 或 git merge --ff 行为一致。

❑ Always Merge

总是产生一个合并提交，即便该变更是目标分支的严格超集。这个行为与 git merge --no-ff 一致，当项目采用 git log --first-parent 来跟踪合入历史时比较有用。

❑ Cherry Pick

总是将变更摘出，并应用到分支的顶端，而忽略其原来的基线。

在这个方式下，Gerrit 会自动在变更被批准时附加一段内容到提交信息，其中包含原变更的 Web 地址。提交者被设置为合入者，作者信息仍保留为原补丁的作者。

❑ Rebase If Necessary

如果变更是目标分支的严格超集，那么分支会被快速推进到该变更。否则的话，该变更被变更基线，然后再将分支快速推进到此变更。

Gerrit 处理合并时，默认只有路径不冲突的时候才可能成功。在被合并的双方都修改了同样的文件时就会产生路径冲突。

❑ Rebase Always

这个方式与 Rebase If Necessary 基本相同，但总是创建一个新的补丁，即便可以执行快速推进。与 Cherry Pick 类似，这个会保留最终被合并提交的各种脚注信息，如 ChangeId 等。

如果 Allow content merge 被使能，Gerrit 会在路径冲突时尝试进行内容合并。

（3）其他选项

项目设置里还有一些其他选项，可能随版本的不同略有差别，具体含义可以参考官方文档中的项目选项列表[⊖]。

3. 分支管理

（1）分支创建

在项目中创建分支有好几种方式：

- 在 Web 界面的 Projects → List → {project} → Branches 里创建。
- 通过 Create Branch 这个 REST 接口创建。
- 通过 create-branch 这个 SSH 命令创建。
- 通过 git 客户端将提交推送到一个不存在的分支。

创建分支需要拥有 Create Reference 访问权限。此外，项目所有者和 Gerrit 管理员可以通过 Web 界面或者 REST 接口创建分支，不受 Create Reference 权限的限制。

在使用 Web 界面、REST 接口或者 SSH 命令这三种方式创建分支时，分支只能建立在已经存在于仓库的提交上。如果分支名不是以 refs/ 开头的，那么会被自动加上 refs/heads 前缀。

分支的起始点可以是任意有效的 SHA-1 表达式，只要可以被解析成提交即可，缩略的 SHA-1 不被支持。

我们以 SSH 命令为例，在实验环境中创建一个新的分支：

```
ssh -p 29418 admin@localhost gerrit create-branch tools/gerrit newbranch master
```

上述命令的效果就是在 tools/gerrit 项目上基于 master 分支创建新的分支 newbranch。

（2）分支删除

分支删除也有好几种方式：

- 在 Web 界面的 Projects → List → {project} → Branches 里创建。
- 通过 Delete Branch 这个 REST 接口创建。
- 通过 git 客户端创建：

```
$ git push origin --delete refs/heads/<branch-to-delete>
```

另外一个方法是强制推送空分支到一个已经存在的分支：

```
$ git push --force origin :refs/heads/<branch-to-delete>
```

（3）默认分支

远程仓库的默认分支由它的 HEAD 引用定义。克隆一个仓库时，Git 会为默认分支创建一个本地分支并检出。

⊖ https://gerrit-documentation.storage.googleapis.com/Documentation/2.14.7/project-configuration.html#project_options。

项目所有者可以设置 HEAD：

- 在 Web 界面的 Projects → List → {project} → Branches 里设置。
- 通过 setHEAD 这个 REST 接口设置。

4. 标签管理

Gerrit 中没有提供专门的标签管理接口，创建标签可以通过 Git 命令完成，用户需要具备相应的权限。前面 3.1.4 节提到标签分为两种：轻量型标签在 Git 中的实现与普通分支类似，因此只需要授予 refs/tags/* 的 Create Reference 权限即可；对于带注记的标签，对应的权限是 Create Annotated Tag，如果需要推送的标签是别人创建的（例如同步上游 Git 仓库时），那么还需要具备 Forge Committer Identify 的权限。具体的 Git 命令如下：

基于 stable 分支创建带注记标签：

```
git tag -a -m "Release v1.0" v1.0 stable
```

向 Git 仓库推送标签：

```
git push ssh://USER@HOST:PROT/PROJECT tag v1.0
```

其中 tag v1.0 的效果与 refs/tags/v1.0:refs/tags/v1.0 一样。

删除或者覆盖已经存在的标签则需要针对 refs/tags/* 的 Push 权限，并同时勾选 force 选项。

5. 权限配置

权限配置可以通过 Web 界面进行：

1）在 Web 界面上选择 Projects → List。

2）找到需要管理的项目并点击进入。

3）点击 Access 菜单项。

4）点击 Edit 进入编辑模式。

5）修改完成之后点击 Save Changes 保存，此时你可以输入一些提交信息来记录这次权限变更的原因。如果没有直接修改的权限，保存之后会进入评审流程。

项目的权限被保存在项目 Git 仓库的一个特殊分支下，即 refs/meta/config。这个分支里有一个 project.config 文件保存了访问权限信息。如果熟悉配置文件的语法，你也可以直接修改这个文件来维护项目权限。样例文件如代码清单 3-2 所示。

代码清单 3-2　Gerrit 权限配置

```
[project]
description = Rights inherited by all other projects
[access "refs/*"]
read = group Administrators
[access "refs/heads/*"]
label-Your-Label-Here = -1..+1 group Administrators
```

```
[capability]
administrateServer = group Administrators
[receive]
requireContributorAgreement = true
[label "Your-Label-Here"]
function = MaxWithBlock
value = -1 Your -1 Description
value = 0 Your No score Description
value = +1 Your +1 Description
```

这个文件的格式与 Git 配置文件一致，将选项分成了不同的区。

- project 配置块只出现一次，保存项目的基本描述。
- access 配置块需要指明匹配模式，配置了各类分支的访问权限，一个匹配模式只能配置一个 access 配置块。
- capability 配置块只出现在 All-Projects 仓库下，保存了适用于全局的一些配置。
- receive 配置块对于每个项目只出现一次，配置了 Gerrit 接收补丁的条件，如上述示例中要求用户必须同意贡献者协议，否则所提的补丁将不被接受。
- label 配置块可以在一个项目里出现多次，除了内建的 Code-Review 和 Verified，你可以增加一些自定义的标签，如 OpenStack 社区里增加了 Workflow 标签用于触发代码合入流程。所有标签的功能、取值和描述都是可以自定义的。

默认情况下，配置文件所在的 refs/meta/config 分支在克隆 Git 仓库时不会被取到本地，我们需要通过 Git 命令显式获取，如代码清单 3-3 所示。

代码清单 3-3　通过 Git 获取配置文件

```
$ git fetch gerrit refs/meta/config
remote: Counting objects: 3, done
remote: Finding sources: 100% (3/3)
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From ssh://localhost:29418/tools/gerrit
* branch  refs/meta/config -> FETCH_HEAD
$ git checkout -b config FETCH_HEAD
Switched to a new branch 'config'
```

上述命令首先获取远端的 refs/meta/config 分支，默认保存在 FETCH_HEAD 下，然后检出该引用到本地的 config 分支。完成修改后即可将变更推送回远端：

```
git push origin config:refs/meta/config
```

3.2.4　CI/CD 系统对接

至此，我们对 Git 和 Gerrit 的基本概念和架构有了初步对了解，为了让提交的代码能够触发 CI/CD 任务，我们还需要完成与 CI/CD 系统的对接，比较常见的有 Jenkins 和 Zuul。我们会在第 4 章和第 5 章分别详细介绍，此处先简单介绍下 Gerrit 这部分的配置。

1. 对接持续集成系统（Jenkins）

Gerrit 可以通过插件[⊖]与 Jenkins 对接，实现代码提交后自动触发构建过程。

1）在 Gerrit 中为 Jenkins 创建一个用户，并为其设置 SSH 密钥。

2）在 Web 界面的 Admin → Groups → Non-Iteractive Users 里加入该用户。

3）在 Web 界面的 Admin → Projects → … → Access → Edit 里配置权限。

配置样例如代码清单 3-4 所示。

代码清单 3-4　Gerrit 对接 CI 权限配置

```
Reference: refs/*
    Read: ALLOW for Non-Interactive Users
Reference: refs/heads/*
    Label Code-Review: -1, +1 for Non-Interactive Users
    Label Verified: -1, +1 for Non-Interactive Users
```

具体可以参考 Gerrit 文档中关于访问控制[⊜]部分的内容。

在 Jenkins 中，需要配置 Gerrit 服务的访问方式以及触发条件。这些触发条件都对应于 Gerrit 中的特定事件，如：

- Draft Published：变更从 draft 状态变为 new。
- Patchset Created：新的补丁集被添加到某一变更，这个是最常用的触发方式之一。
- Change Merged：变更被合入 Gerrit 服务器，这个通常用于触发发布流程。
- Comment Added：评论被添加到某一变更，这个通常用于手动触发构建过程，如 OpenStack 里的 recheck。
- Ref Updated：当一个 Git 引用被更新时，除了通过正常的评审流程合入代码之外，增加 tag、创建分支等操作也会触发此事件。

2. 对接门控系统（Zuul）

在 OpenStack 的 CI/CD 系统中，Gerrit 并非直接与 Jenkins 对接，而是引入了另外一个系统 Zuul 来进行项目管控。简单来说，Zuul 会侦听 Gerrit 的事件流，然后触发 Jenkins 上的构建任务。具体请参见第 5 章。

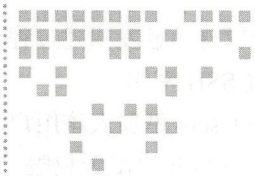
3.3　本章小结

在本章中，我们介绍了 Git 和 Gerrit 的基本功能和它们在 CI/CD 中扮演的角色。

Gerrit 对接开发者和 Git 仓库，提供了身份验证和访问控制功能，保存评审历史，并输出事件流以触发 CI/CD 构建任务。它是整个自动化流水线的起点，吹响了 DevOps 的号角。

⊖ https://wiki.jenkins.io/display/JENKINS/Gerrit + Trigger。

⊜ https://gerrit-review.googlesource.com/Documentation/access-control.html。



Chapter 4 第 4 章

持续集成系统（Jenkins）

所谓持续集成，就是指频繁地（一天多次甚至几十次）将代码集成到主干。一个完整的持续集成系统必须包括：

- 代码存储库：即版本控制软件来保障代码的可维护性，同时作为构建过程的素材库。
- 自动构建过程：包括自动编译、分发、部署和测试等。
- 持续集成服务器：实时监控代码存储库的活动，并启动自动构建过程。

OpenStack 使用 Jenkins 来作为持续集成服务器。

4.1 Jenkins 介绍

Jenkins 的前身是 Hudson，是 Sun 公司于 2004 年夏天开发的，2005 年 2 月底开源并发布了第一个版本。Hudson 在 2008 年左右就已经基本取代了 CruiseControl 和其他开源的构建服务器，并在 2008 年 5 月的 JavaOne 大会上，获得开发人员解决方案分类的 Duke 选择奖。

2009 年，Oracle 收购了 Sun，之后 Oracle 把 Hudson 名字变成注册商标并开始作为商业软件开发。2011 年年初，开发社区通过投票将项目改名为 Jenkins，持续维护。目前，在 GitHub 上的 Jenkins 组织有 600 多个项目成员和 1900 多个公共库。

4.1.1 Jenkins 是什么

Jenkins 是一个用 Java 编写的开源的广泛用于持续构建的可视化 Web 集成工具。它支持各种项目（各种开发语言如 Net、Ruby、PHP、Python 等）的自动化编译、打包、分发和

部署。完全兼容 ant、maven、gradle 等第三方构建工具，同时能与 SVN、CVS、Git 等软件配置管理工具（SCM）无缝集成，也直接支持第三方源码托管系统，如 GitHub、GitLab、BitBucket 等。它具有如下优点：

- 易用性：支持 Windows、Linux、Mac 等操作系统，提供可视化 Web 配置管理。
- 可扩展性：扩展插件涵盖了版本控制、构建工具、UI 自定义、事件通知等。
- 活跃的社区：大量的用户和插件，足见应用的广泛性和社区活跃度。
- 稳定的版本支持：每周发布新的版本，每 12 周发布一个稳定的 Long-Term Support（LTS）版本。

4.1.2　Jenkins 工作原理

Jenkins 收到触发构建的命令后，先选择一个合适的 Slave，并将待构建代码拷贝一份到 Slave，然后根据预定义的自动构建脚本执行一系列任务。Jenkins 触发构建有多种，最常用的有如下几种：

- 版本控制系统中有新的变更或者补丁
- 类似 Cron 的定时任务
- 其他构建完成时
- 收到特定的 URL 请求

1. 典型的持续集成系统

下面以 Jenkins + Gerrit 组成的系统为例来说明一个典型的持续集成系统（如图 4-1 所示）是如何工作的。

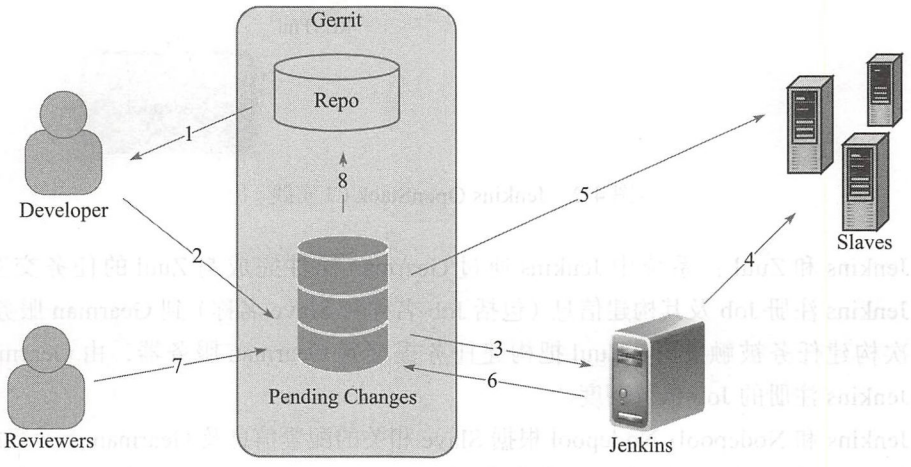


图 4-1　Jenkins-Gerrit 持续集成系统

1）开发者从代码库中检出源代码，进行新功能开发、修改 bug 或者更新文档等变更。

2）开发者提交新的变更到 Gerrit。

3）Gerrit 发送代码库变更消息给 Jenkins。

4）Jenkins 收到代码库变更消息后，根据构建任务配置选择合适的 Slave（slave 选择规则请参见 4.1.6 节中的 labels 描述）。

5）Jenkins 远程控制 Slave 从代码库 Gerrit 中检出变更代码，启动变更验证。

6）Jenkins 把变更验证结果通过投票形式通知 Gerrit。

7）其他开发人员评审变更，并投票是否允许合入。

8）如果此变更被 Jenkins 及其他开发人员都投票通过，则 Gerrit 把此次变更合入代码库。

2. OpenStack Jenkins 实践

在 OpenStack CI 系统中，构建任务通过 Zuul 触发调度，而 Slave 则是通过 Nodepool 进行动态的注册和注销。组件之间的交互关系如图 4-2 所示。

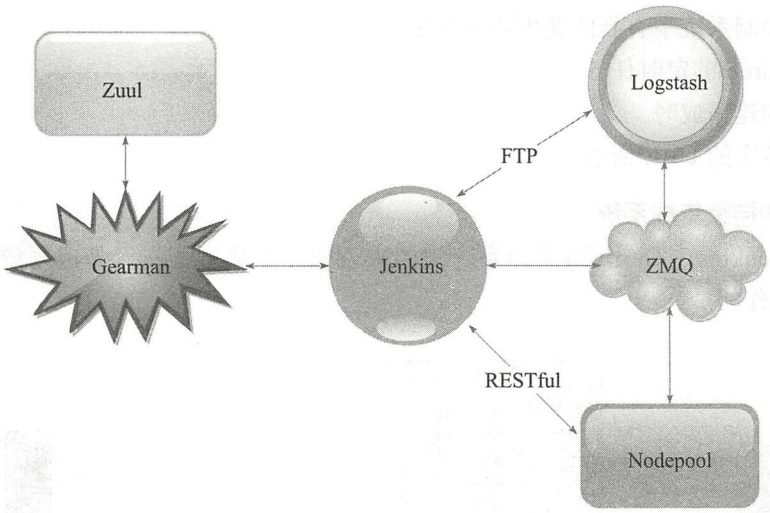


图 4-2　Jenkins OpenStack CI 实践

- Jenkins 和 Zuul：系统中 Jenkins 通过 Gearman 插件完成与 Zuul 的任务交互功能，Jenkins 注册 Job 及其构建信息（包括 Job 名称和 Slave 名称）到 Gearman 服务器，每次构建任务被触发时，Zuul 把构建任务提交到 Gearman 服务器，由 Gearman 根据 Jenkins 注册的 Job 执行调度。
- Jenkins 和 Nodepool：Nodepool 根据 Slave 相关的配置信息及 Gearman 上运行的 Slave 状态（数目及类型等）计算出需要创建的 Slave，并通过 RESTful 接口注册给 Jenkins；另一方面每次构建完成之后，Jenkins 通过 ZMQ 发布构建完成消息，Nodepool 收到消息后，通过 RESTful 接口注销 Slave。

❑ Jenkins 和 Logstash：每次构建完成之后，Jenkins 通过 ZMQ 发布构建完成消息，Logstash 收到消息之后，进行构建日志的采集。

本章主要通过典型的持续集成系统（Gerrit + Jenkins）对 Jenkins 的应用进行描述，至于 Jenkins 在 OpenStack CI 系统中的实践请参见第 10 章。

4.1.3　部署 Jenkins

1. 安装 Jenkins

Jenkins 是基于 Java 开发并开源的自动化持续集成系统，所以在任何支持 Java 的操作系统或平台中都可以安装 Jenkins。它支持 War 包和容器安装方式，具体的安装方法见下文描述。

（1）WAR 文件安装

通过 War 包安装的过程如下：

1）下载指定或者最新的稳定的 War 包到待部署主机的指定目录。

2）打开控制终端并进入到下载目录。

3）执行 java-jar jenkins.war 命令启动 Jenkins。

4）通过 http://jenkins.cibook.oz:8080 链接，访问 Jenkins。

5）等待 Jenkins 启动完成之后，在访问页面上自动进入启动后引导程序。

如果需要修改 Jenkins 监听端口，可以通过设置 --httpPort 启动参数来完成，比如通过运行 java-jar jenkins.war -httpPort=8711，Jenkins 就监听在端口 8711 上了，这时候需要通过 http://jenkins.cibook.oz:8711 访问 Jenkins 界面。

（2）Docker 安装

Jenkins 也提供了容器化的使用方式。通过容器启动 Jenkins 步骤如下：

1）打开控制终端。

2）通过代码清单 4-1 中的命令下载并启动 Jenkins 容器。

代码清单 4-1　启动 Jenkins 容器

```
docker run \
    -u root \
    --rm \
    -d \
    -p 8080:8080 \ #a
    -p 50000:50000 \ #b
    -v jenkins-data:/var/jenkins_home \ #c
    -v /var/run/docker.sock:/var/run/docker.sock \
    jenkins/jenkins:latest
```

其中：

❑ -p jenkins_port:8080：把容器内部 Jenkins 的监听端口 8080 映射到主机的 jenkins_

非卖品，仅供非商业用途或交流学习使用

port 端口，可以通过修改 jenkins_port 来修改主机端口。比如，如果写成 -p 49001:8080，Jenkins 的监听端口就变成了 49001。

- -p slave_port:50000：Slave 和 Jenkins 通信的 TCP 端口。同上，此端口也是可以被修改的。
- -v jenkins_data:/var/jenkins_home：可选，但是强烈建议使用；通过 Docker 卷的概念把 Jenkins 配置数据映射到主机的 jenkins_data 目录。如果不加入此映射，每次 Jenkins 重启都会是一个新的实例，老的数据将不复存在。

3）通过 http://jenkins.cibook.oz:8080 访问 Jenkins 页面。

4）等待 Jenkins 启动完成之后，就进入启动后引导程序。

2. 启动后引导程序

在 Jenkins 安装成功之后，就进入启动后引导程序流程，这一流程通过一系列的一次性步骤完成解锁 Jenkins、定制化插件及产生第一个管理员用户，详细步骤如下所述。

（1）解锁 Jenkins

当 Jenkins 安装启动之后，第一次被访问的时候，会要求用户使用 Jenkins 自动生成的密码来解锁它，这样做是从安全的角度出发迫使用户新创一个用户名密码，如图 4-3 所示。

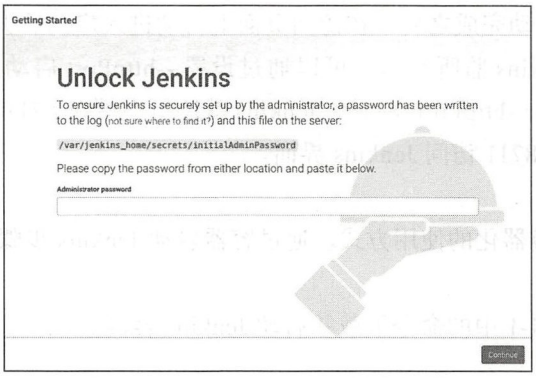


图 4-3　解锁 Jenkins

如果是以 War 包方式启动的，这个密码可以从控制终端的打印日志中获取，也可以根据页面的提示到存储文件中获取（linux 默认路径：/var/jenkins_home/secrets/initialAdminPassword）。如果 Jenkins 是通过 Docker 启动的，可以到 Docker 日志中（通过 docker log）获取。

（2）定制插件

Jenkins 解锁之后即进入插件定制化页面，这步主要是选择初始安装插件，系统提供了两个选项：

- Install suggested plugins：安装默认的插件，Jenkins 针对常用应用场景，提供了一系列默认的安装插件。在不知道哪些插件需要安装的情况下，可以选择此选项，后

非卖品，仅供非商业用途或交流学习使用

续可以根据需要再增加或者删除部分插件。

- Select plugins to install：安装自定义插件，如果你非常明确需要安装哪些插件，默认的插件又不能满足需求，可以选择此选项。

开始安装之后，会有一个进度条显示当前正在安装的插件及安装进度，整个过程会持续几分钟。

（3）创建管理员用户

在插件安装完成之后，系统会进入创建第一个管理员用户页面，在这个页面里填入管理员用户的用户名和密码，之后即可用此用户访问并管理 Jenkins 了。当然，为了安全考虑，一般创建一些用户并根据角色和权限进行管理，详情请参见 4.1.5 小节。

4.1.4　插件管理

Jenkins 的插件除了在前面介绍的通过启动后引导程序安装之外，还可以在启动完成之后在使用过程中通过 Manage Jenkins → Manage Plugins 页面执行安装、删除、更新等管理操作。

1. 安装插件

Jenkins 插件支持在线安装，也支持离线安装。通过 Manage Jenkins → Manage Plugins 进入插件安装页面。

图 4-4 以安装 Gerrit Trigger 为例来介绍在线安装方式。

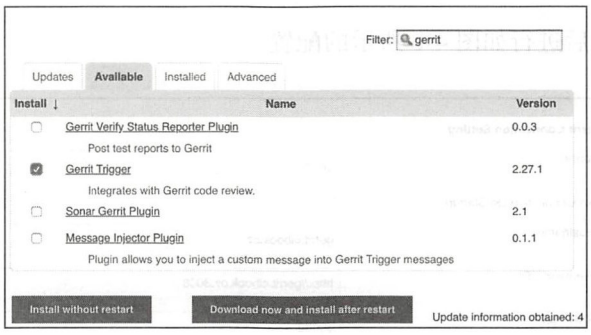


图 4-4　安装插件

通过 Available 页面的 Filter 匹配待安装的插件，选择好插件之后点击 Install without restart，这个插件就安装完成了。

离线安装是指先下载好插件的 hpi 文件，在 Advanced 页面通过上传 hpi 文件进行安装的方式，这里不再详加说明，感兴趣的读者可以参考官方文档[⊖]。需要注意的是，这种方式安装的插件需要重启 Jenkins 才能生效。

⊖ https://jenkins.io/doc/book/managing/plugins/#Advanced%20installation。

非卖品，仅供非商业用途或交流学习使用

2. 配置插件

许多插件安装完成之后，必须进行配置才能使用，其相关插件的配置基本都是通过 Manage Jenkins → Configure System 来实现的，下面介绍几个常用插件的配置。

（1）Gerrit Trigger

Gerrit 的配置涉及 Gerrit 本身的配置和 Jenkins 上 Gerrit Trigger 插件的配置两部分，Gerrit 的配置请参见第 3 章。

Gerrit Trigger 插件的配置路径与其他插件稍稍不同，是通过 Manage Jenkins → Gerrit-Trigger 进入的。如果是第一次配置，点击页面左上角的 Add New Server，如果对已经加入的 Gerrit 配置进行修改，则点击图 4-5 中 Edit 下面的图标。

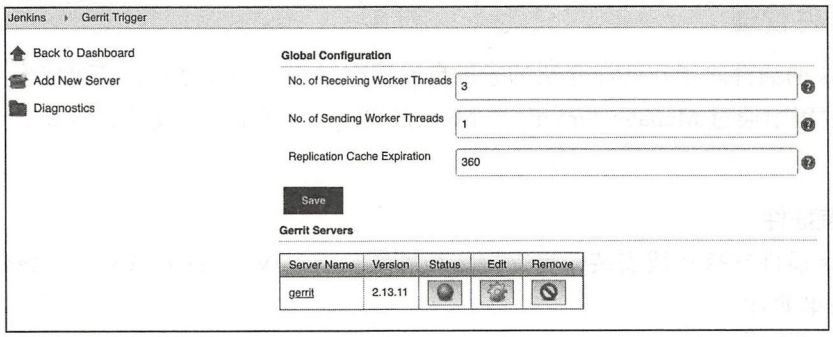


图 4-5　Gerrit Servers 页面

进入配置页面之后进行如图 4-6 所示的配置。

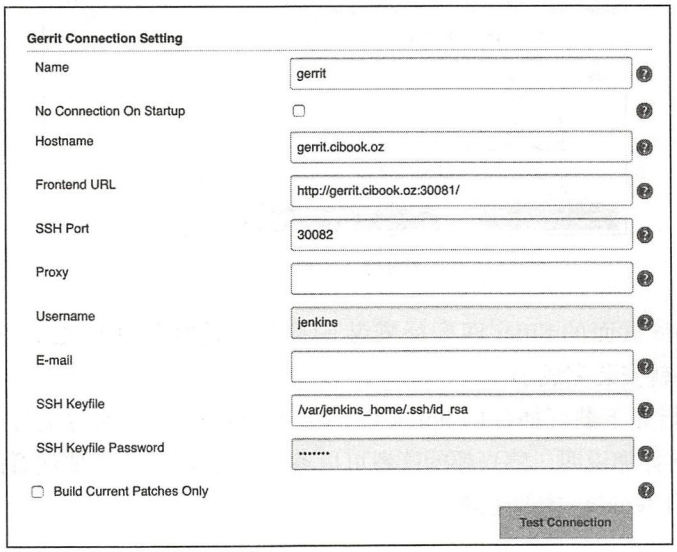


图 4-6　配置 Gerrit Server

非卖品，仅供非商业用途或交流学习使用

主要配置项含义如下：

- Name：Gerrit 服务器的名字。
- Hostname：Gerrit 服务器的主机名或者 IP 地址。
- Frontend URL：访问 Gerrit 页面的 URL。
- SSH Port：SSH 方式下载代码使用的端口号，默认是 29418，在此示例环境中使用的是 30082 端口。
- Username 和 SSH Keyftle：SSH 访问 Gerrit 的用户名和密钥。
- Test Connection：配置完成后点击此按钮检查配置是否正确，如果正确则会显示 Success。

此外，为了实时返回构建过程及自动返回构建结果给 Gerrit，还需要配置 Gerrit Reporting Values（如图 4-7 所示）和 Gerrit Verifted Commands（如图 4-8 所示）。

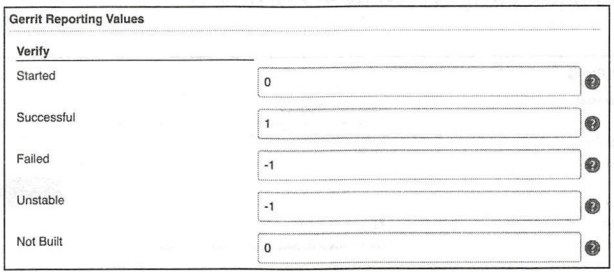


图 4-7　Gerrit Reporting Values

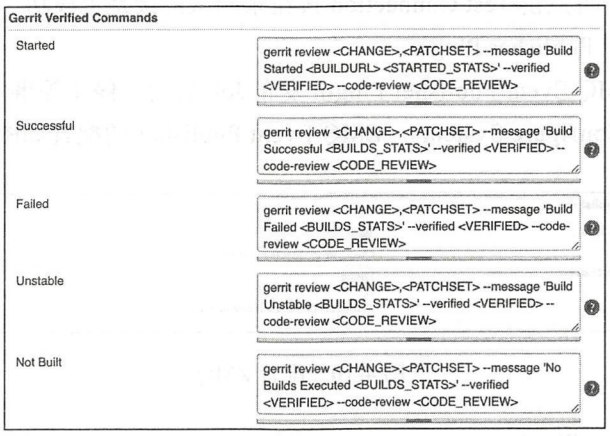


图 4-8　Gerrit Verified Commands

经过 Gerrit Reporting Values 和 Gerrit Verifted Commands 两项配置，可达到如下效果：

1）构建开始：Jenkins 向 Gerrit 发送 Build Started 和 Verifted=0 的消息，Gerrit 收到消息后添加一条构建开始的评审记录，并清除 verified 值。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

2）构建结束并成功：Jenkins 向 Gerrit 发送 Build Successful 和 Verifted + 1 的消息，Gerrit 收到消息后添加一条构建成功的评审记录，并置 Gerrit 上 Verified 为 + 1。

3）构建结束并失败：Jenkins 向 Gerrit 发送 Build Failed 和 Verifted-1 的消息，Gerrit 收到消息后添加一条构建失败的评审记录，并置 Gerrit 上 Verified 为 −1。

4）构建不稳定：Jenkins 向 Gerrit 发送 Build Unstable 和 Verifted-1 的消息，Gerrit 收到消息后添加一条构建不稳定的评审记录，并置 Gerrit 上 Verified 为 −1。

5）如果构建没有启动：Jenkins 不会向 Gerrit 发送消息。

配置完成，点击 save，就会自动返回到图 4-5 所在的页面，这时点击 Status 栏的球形图标，如果球的颜色标志成绿色，则证明此 Gerrit 服务器对接成功，可以使用了。

（2）Gearman Plugin

Gearman Plugin 是 Jenkins 和 Gearman 进行通信的桥梁，在 Manage Jenkins → Configure System → Gearman Plugin Config 按照如图 4-9 所示配置。

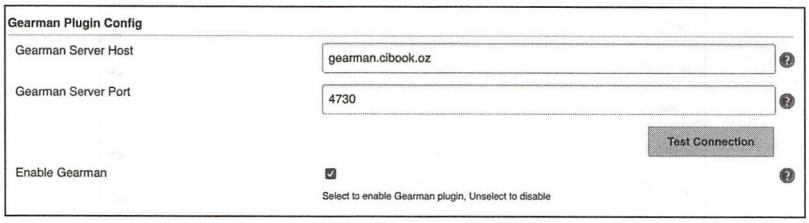


图 4-9　配置 Gearman

配置完成之后，可点击 Test Connection 按钮来测试链接是否成功。

（3）ZMQ Event Publisher Plugin

Jenkins 通过 ZMQ Event Publisher Plugin 发布 Job 启动、停止等事件到 Nodepool，其在 Manage Jenkins → Configure System → ZMQ Event Publisher 的配置如图 4-10 所示。

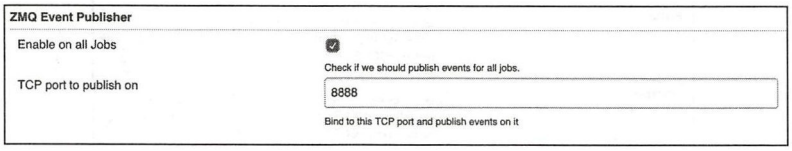


图 4-10　配置 ZMQ

（4）SCP Pulisher Plugin

OpenStack CI 系统中持续集成的日志通过 SCP 转存到日志服务器中。SCP Publisher Plugin 是 Jenkins 上负责此转存操作的载体，其在 Manage Jenkins → Configure System → SCP repository hosts 的配置如图 4-11 所示。

配置完成之后，可以通过 Test Connection 测试其连通性。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

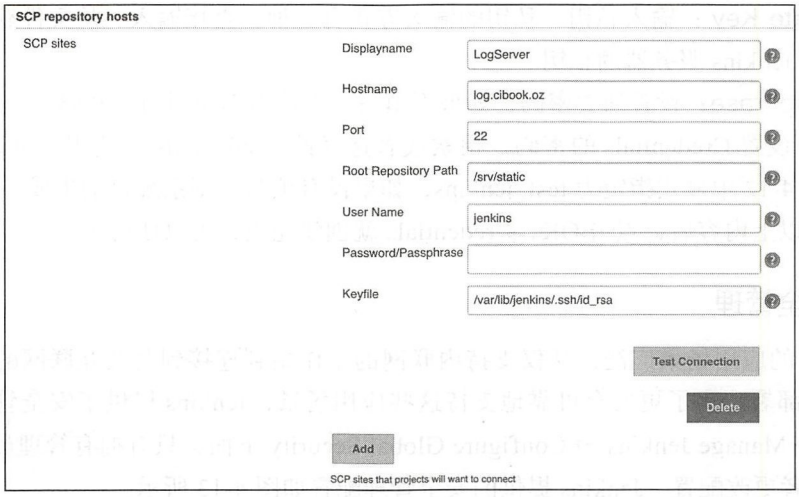


图 4-11　配置 SCP Publisher

（5）SSH Credentials Plugin

OpenStack CI 系统中，Jenkins 和 Slave 的通信安全通过使用 SSH Credentials 来保障，在安装完 SSH Credentials Plugin 之后，点击 Crendentials → System → Global credentials 进入 Global Crendentials 页面，然后点击 Add Credentials 创建新的 Credential，见图 4-12。

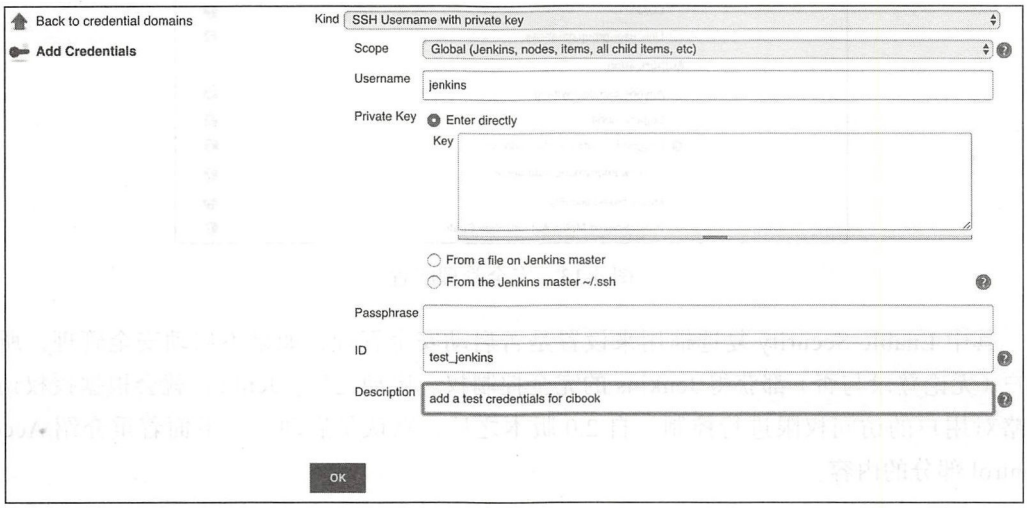


图 4-12　创建 Credentials

- Kind：选择 Credentials 类型，可以是用户名密码格式，也可以是 SSH 密钥格式。
- Scope：选择 Credentials 的使用范围，系统提供了两种：global 全局使用，system 只用于 Jenkins 和 Slave 通信。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- Private Key：输入私钥，私钥的输入方式有三种，直接输入、从文件获取或者直接使用 Jenkins 服务器的私钥。
- Passphrase：设置通关密码，如果不填写，则认为不使用通关密码。
- ID：设置 Credentials 的密码，检索或者选择此 Credentials 时使用，可以显式指定，如图 4-12 中显式指定为 test_jenkins，如果没有填写，则系统自动生成一个 UUID。

填写完以上内容后，点击 OK，Credentials 就创建完成，可以使用了。

4.1.5 安全管理

Jenkins 的应用场景广泛，从仅支持内联网的工作站到连接到公共互联网的高性能服务器上都可以部署。为了更安全可靠地支持这些应用场景，Jenkins 提供了安全管理功能，放在 jenkins → Manage Jenkins → Configure Global Security 下面，只有拥有管理员权限的用户可以查看或者更改配置。Jenkins 提供的安全管理配置如图 4-13 所示。

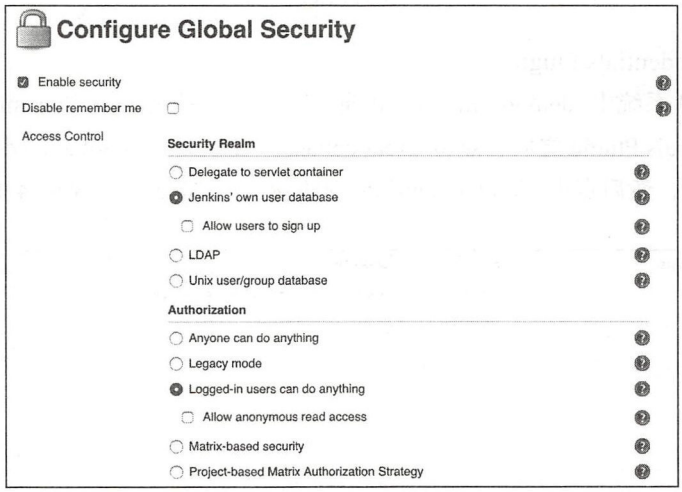


图 4-13 安全管理配置

其中 Enable Security 复选框用来设置是否启动安全管理，如果不启动安全管理，所有用户（无论登录与否）都获得 Jenkins 的完全控制权。启动之后，Jenkins 就会根据授权访问策略对用户的访问权限进行控制。自 2.0 版本之后，默认是启动的。下面着重介绍 Access Control 部分的内容。

1. Security Realm

Security Reaml，即身份认证（Authentication），配置用户的认证方式，Jenkins 默认支持以下几种认证方式：

- Delegate to servlet container：委托运行 Jenkins 的 Servlet 容器进行身份认证，如 Jetty。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- Jenkins'own user database：使用 Jenkins 内建的用户数据库，自 2.0 版本以后的默认配置，在小型系统中推荐使用。
- LDAP：委托外置的 LDAP 服务器进行身份认证，在较大的或者已经提供了 LDAP 认证的公司或者组织里推荐使用。需要注意的是，这个功能需要安装 LDAP plugin。
- Unix user/group database：委托 UNIX 操作系统级的用户数据库进行身份验证。在这种方式下，用户使用 Jenkins 所在 UNIX 主机的用户名和密码进行登录。

2. Authorization

即授权访问，用来控制用户可以访问哪些资源，Jenkins 默认支持如下授权访问方式：

- Anyone can do anything：不实施任何授权访问控制，Jenkins 的权限对所有用户开放，包括没有登录的匿名用户，建议仅用于测试环境。
- Legacy mode：被赋予管理员角色的用户，获得 Jenkins 的所有控制权限，否则仅有读权限，建议仅用于测试环境。
- Logged in users can do anything 登录用户获得 Jenkins 的所有控制权限，未登录用户，如果选中 Allow anonymous read access 则获得读权限。
- Matrix-based security 通过用户权限管理表设置用户或者用户组的访问权限，对于大多数 Jenkins 用户来说，此种授权访问方式能够提供足够的安全控制和灵活性。所以，没有其他特殊要求的生产环境，可以使用此种方式。配置方式如图 4-14 所示。

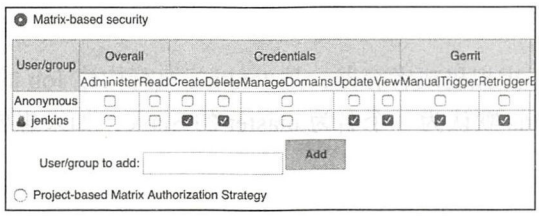


图 4-14　Matrix-based security

图 4-14 中的列表示权限控制项，需要注意的是，图中只显示了部分配置，实际中配置项很多，而且随着插件的增多，配置项也会变多，比如其中的 Gerrit 权限项就是由于安装了 Gerrit Trigger 插件引入的。图中的行配置了用户或者用户组的访问权限。

注解　在 Matrix-based security 的模式下，用户权限是进行叠加的，也就是说如果一个用户同时属于用户组 A、B 和 C，那么这个用户的权限为用户组 A、B、C 及匿名用户的合集。

- Project-based Matrix Authorization Strategy：此权限控制方式是 Matrix-based security 的扩展，它在用户和用户组管理的基础上又加入了各个项目的访问控制（在 Job 页面进行配置），比方说用户 Joe 有项目 A 和 B 的访问权限，但是却没有项目 D

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

的访问权限。具体配置使用方式此处不做详细介绍，有兴趣的用户请自行查阅官方文档[⊖]。

3. 用户管理

Jenkins 通过 jenkins → Manage Jenkins → Manage Users 进行用户的增加或者删除操作。用户的增加和删除操作很简单，此处不再说明。

需要说明一点的是，Jenkins 自 1.426 版本开始，为 RESTful 访问提供了 API token 来替代密码进行身份认证。用户可以在个人设置页面查看和更改 API token，如图 4-15 所示。

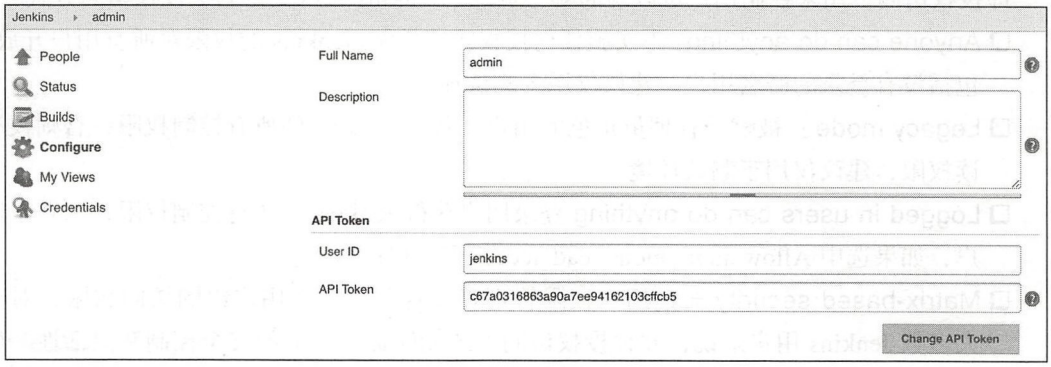


图 4-15 API Token

4.1.6 创建 Slave

Slave 可以是主机、虚拟机或者容器，是 Jenkins 执行构建 Job 的载体，没有 Slave，Job 就不能够被执行。Jenkins 默认为一个名为 master 的 Slave，如果想要创建新的 Slave，可以简单分为以下两步：

1）配置 Slave。

2）把 Slave 挂载在 Jenkins 上。

下面我们以创建一个 SSH 密钥访问型的 Slave 为例来说明创建方式。注意，这种类型的 Slave 需要安装 SSH Slave Plugin。

1. 配置 Slave

Slave 的配置，简单来说就是安装 Java JDK 和配置 Jenkins 信任的 SSH 公钥，配置命令如代码清单 4-2 所示。

代码清单 4-2 配置 Slave

```
apt-get install openjdk-7-jdk
cat id_rsa.pub > ~/.ssh/authorized_keys
chmod 400 ~/.ssh/authorized_keys
```

⊖ https://jenkins.io/doc/book/managing/security/#authorization。

仅供非商业用途或交流学习使用

注解　这里的公钥 id_rsa.pub 必须与 4.1.4 节中创建的用于 Jenkins 和 Slave 进行 SSH 通信的安全证书的密钥是一对。

2. 挂载 Slave

在 Jenkins 页面上选择 Manage Jenkins → Manage Nodes → New Node，输入 Slave 的名字，并选择类型为 Permanent Agent，点击确认之后进入如图 4-16 所示配置页面。

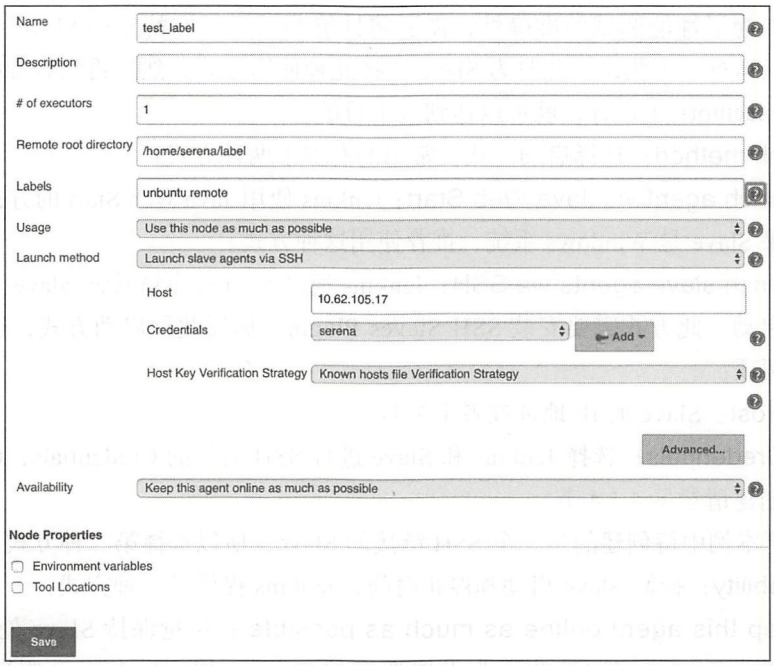


图 4-16　挂载 Slave

在此页面做如下配置：

- Name：Slave 的名字，在同一个 Jenkins 上，Slave 的名字必须是唯一的。
- # of executors：Slave 上可以同时执行的 Job 数目，一个 Executor 承载一个 Job，如果此值大于 1，则只有当所有的 Executor 都消耗完，Slave 的生命周期才算结束。如无特殊要求，此值一般默认设置为 1。
- Remote root directory：构建执行时 Slave 上的工作目录，在此设定下，构建时 Slave 上的运行目录为：$WORKSPACE=/home/serena/label/workspace/{JOBNAME}。
- Labels：Slave 的标签。标签的作用是为 Slave 分组，供 Job 调度时选择 Slave 用，比如同是 Ubuntu 系统的 Slave 都打上 ubuntu 标签就组成一个 Ubuntu 组，然后 Job 配置的时候可以通过设置 ubuntu 标签绑定某些特定 Job 只运行在安装 Ubuntu 系统的 Slave 上（Job 绑定 Slave 的方法请参见 4.1.7 和 4.2.4 两节）。一个 Slave 可以不设

仅供非商业用途或交流学习使用

置标签，也可以设置一个或多个，多个标签用空格隔开。比如，图 4-16 中就设置了 ubuntu 和 remote 两个标签。

- Usage：Slave 使用方式，分为两种：
  - Use this node as much as possible：默认配置，只要构建可以使用这个 Slave，Jenkins 就会尽量调度 Job 到这个 Slave 上。
  - Only build jobs with label expressions matching this node：只有匹配此 Slave 名称或者标签的 Job 才能运行在这个 Slave 上。这种模式可以为特定 Job 预留 Slave，比如为了性能测试的准确性，需要将性能测试运行在特定的 Slave 上，并保证一次只运行一个测试，此时为 Slave 选择此种使用方式，然后再结合 labels=perf 和 # of executors=1 设置，就可以达到这个目的。
- Launch method：选择启动方式，常用的有如下两种方式：
  - Launch agent via Java Web Start：Jenkins 使用 Java Web Start 的方式启动 Slave，如果 Slave 是 Windows 系统，推荐使用这种方式。
  - Launch slave agents via SSH：Jenkins 通过 SSH 安全链接向 Slave 发送命令的方式启动。此方式需要安装 SSH Slaves Plugin。选择此种启动方式，还需要配置如下两项：
    - Host：Slave 的 IP 地址或者主机名。
    - Credentials：选择 Jenkins 和 Slave 进行 SSH 通信的 Credentials，Credentials 的创建请参见 4.1.4 节。

  由于本例中待创建的是一个 SSH 链接的 Slave，所以选择第二种方式。
- Availability：控制 Slave 启动和停止时间，Jenkins 提供了三种方式：
  - Keep this agent online as much as possible：尽量保持 Slave 处于在线状态。如果由于网络问题等非人为设置造成的 Slave 下线，Jenkins 会周期性地尝试重启它。
  - Take this agent online and offline at speciftc times：Jenkins 根据设定的启动时间和时长来启动和停止 Slave。如果 Slave 在应该在线的时间下线了，Jenkins 会周期性地尝试重启它。如果下线时间到，但是构建 Job 还在运行，Jenkins 会在构建 Job 结束后再停止 Slave。
  - Take this agent online when in demand，and offline when idle：有构建 Job 调度到此 Slave 上时启动 Slave，如果一段时间内 Slave 没有接到构建任务，Jenkins 就会停止 Slave。

以上小节中说的周期性重启尝试和启动时间的设置都基于 cron 系统时间的指令格式。

仅供非商业用途或交流学习使用

其余保持默认配置即可，点击确认。至此，一个 Slave 即创建完成。可以到 Jenkins → Manage Jenkins → Manage Nodes 页面查看 Slave 的状态，如图 4-17 中 master 的状态为正常可以使用，而 test_label 为连接或者启动失败，不能被调度使用，此时就需要排查启动失败原因，直到启动成功。

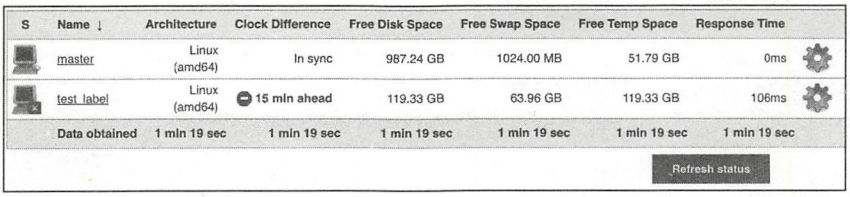


图 4-17　Slave 状态

4.1.7　创建 Job

Jenkins 安装配置完成之后，就可以创建 Job 来进行自动化测试和部署了。系统默认只支持 FreeStyle 类型，如果想使用其他的 Job 类型（如 multijob、pipeline 等），需要自行安装相应的插件。下面就通过创建一个 FreeStyle 类型的 Job 来简述 Job 创建流程。

在 Jenkins 页面的左侧面板中点击 New Items，在 Enter an item name 框中输入 test_freestyle，选择 Freestyle Project，然后点击 OK 按钮，至此一个 Job 框架创建完成。一个 Job 需要的典型配置项如图 4-18 所示。

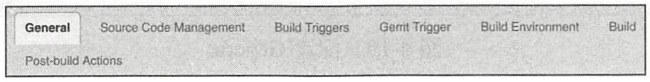


图 4-18　典型 Job 配置项

由于编程语言、代码库、触发条件及构建完成后的处理等不同，需要安装的插件也会不同，从而 Job 的配置也会千差万别。本书不会对所有的配置项都做详细说明，需要的用户可以根据实际需要挑选相关插件，并通过 Jenkins 页面提供的在线帮助及插件的使用说明文档来详细了解。下面仅就 test_freestyle 用到的配置项做详细介绍，并对一些常用或者典型的配置做简略说明。

1. General

设置 Job 运行的参数和属性，如图 4-19 所示。

常用的配置如下：

- Project name：Job 名称。
- Discard old builds：构建日志保存时长。
  - Days to keep builds：最长保存多少天的日志。
  - Max # of builds to keep：最多保存多少条记录。

仅供非商业用途或交流学习使用

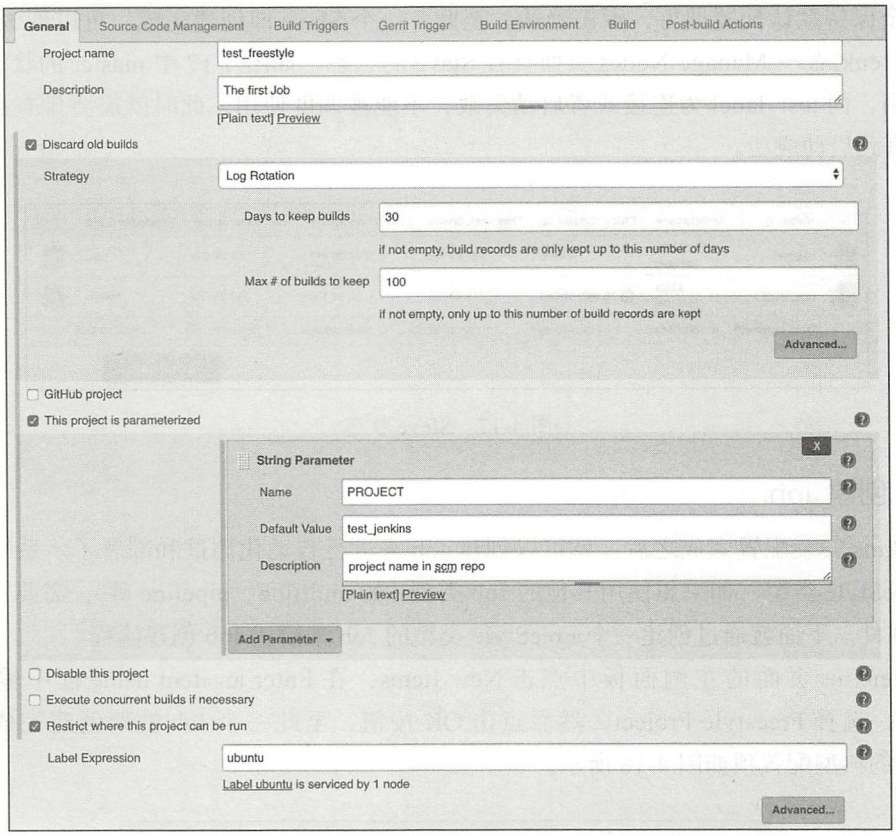


图 4-19 配置 General

- GitHub project：项目的 GitHub URL。
- This project is parameterized：Job 构建需要的参数，如定义构建分支，项目的 URL 等。Jenkins 通过参数设置为 Job 的构建定义一个或者多个参数。这些参数以键值对的方式作为环境变量传递给构建脚本。Jenkins 提供了多种类型的构建参数，如表 4-1 所示。

表 4-1 常用 Parameters

Parameters	功能	Plugin
Bool Parameter	定义布尔变量（true/false）	None
Choice Parameter	定义枚举变量	None
Credentials Parameter	选择 credentials	Credentials Plugin
File Parameter	指定上传文件路径	None
Label	设置执行构建的 Slave 标签	None
Node	列出可执行构建的所有 Slave	None
String Parameter	定义字符串变量	None

- Disable this project：如果此配置项被选定，这个 Job 将不会被触发。
- Execute concurrent builds if necessary：控制项目的多个构建是否可以并行执行。如果选定此选项，且 Slave 的 executors 配置大于 1，则一次最多可以有 executors 个构建 Job 并行执行。
- Restrict where this project can be run：根据 Slave 名称或者标签，限定 Job 可执行构建的 Slave。
- Block build when upstream project is building：当上游依赖项目正在构建或者在构建队列中时，阻止 Job 触发新的构建。此时构建任务会被放在构建队列中，等所有依赖项目都构建完成，再执行构建。
- Block build when downstream project is building：当下游子项目正在构建或者在构建队列中时，阻止 Job 触发新的构建。此时构建任务会被放在构建队列中，等所有依赖项目都构建完成，再执行构建。
- Throttle Concurrent Builds：设定项目最多可并行执行的构建数目，需要安装 Throttle Concur-rent Builds Plugin。
- ZMQ Event Publisher：通过 ZMQ 发布 Job 运行事件（启动、停止等），需要安装 ZMQ Event Publisher。

在 test_freestyle 中把待测项目在 Gerrit 中的名称设置为 PROJECT 参数，保持 30 天或者最多 100 条构建记录，并限定这个 Job 只能运行在标签为 ubuntu 的 Slave 上。

2. Source Code Management

选择并配置软件配置管理工具。指定项目的源代码位置，它将 SCM 属性添加到 Job 配置中，Job 接受任意数量的 SCM 定义。定义多 SCM 时，需要安装 Multiple SCMs plugin。

前面已经讲过 Jenkins 支持很多 SCM，每个 SCM 的配置不同，在此我们不一一描述，用到时请自行查阅相关资料。本例中使用 Gerrit 作为代码库，配置如图 4-20 所示。

主要的配置项说明如下：

- Repositories：代码库的访问配置。Gerrit 相关的详细参数说明，请参见第 3 章。
  - Repository URL：代码库的访问链接，也即 git clone 命令使用的链接。
  - Credentials：同代码库链接的 Credentials，配置方法参见第 4.1.4 小节。
  - Name：远端代码库的唯一标识（ID），如果不填写，Jenkins 会自动分配一个，在 git remote 命令中使用。
  - Refspec：本地引用和远端引用的映射，在 git fetch 和 git push 命令中使用。Gerrit Event 触发方式下，此处必须写为 $GERRIT_REFSPEC。
- Branches to build：待构建的分支。
- Additional Behaviours：Git Plugin 是可扩展的，它同专有插件一起来完成 SCM 配置。附加选项有很多，这里不一一说明，如果想要了解每项的含义和用途，请参阅 Jenkins 页面的在线帮助。

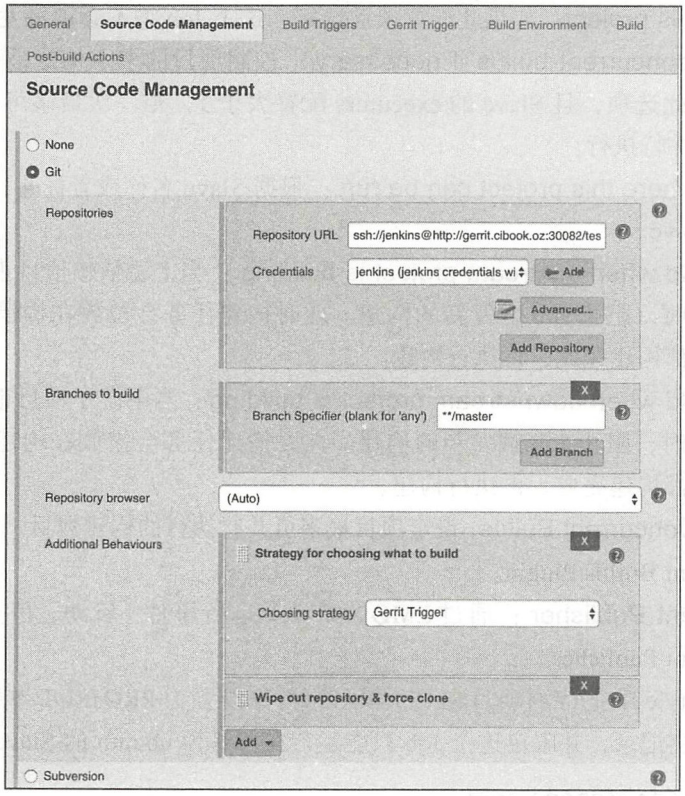


图 4-20　配置 SCM

- Strategy for choosing what to build：本例配置 Gerrit 作为构建触发代码库，必须选择 Gerrit Trigger。
- Wipe out repository & force clone：构建开始前清空工作目录。

3. Build Triggers

设置构建触发条件。如前文所述，构建的触发条件有多种且不是固定不变的，SCM 相关插件的安装会引入新的选项，如 Gerrit Trigger Plugin 的安装会引入 Gerrit Event。Jenkins 常用的触发方式有：

- Trigger builds remotely：通过 Job 的 URL 来触发构建，为了安全考虑，此触发方式建议使用授权 token 做认证，触发构建的 URL 为 JENK-INS_URL/job/JOB_NAME/build?token=TOKEN_NAME 或者 /buildWithParame-ters?token=TOKEN_NAME。
- Build after other projects are built：在上游项目构建结束后触发，根据需要可设置触发条件是上游项目构建成功、失败或者只要构建结束就触发，需要安装 Build-ResultTrigger Plugin。
- Build periodically：周期性触发构建，时间设置基于 cron 系统时间的指令格式。

- Poll SCM：周期性轮询代码库，如果有代码变更则触发，时间设置基于 cron 系统时间的指令格式。
- GitHub hook trigger for GITScm polling：GitHub 代码库更新触发，需要安装插件 GitHub Plugin。
- Build when a change is pushed to GitLab：GitLab 代码库更新触发，需要安装插件 GitLab Plugin。
- Gerrit event：Gerrit 代码库更新触发，需要安装插件 Gerrit Trigger Plugin。

在 Build Triggers 选择 Gerrit event 触发之后就会进入 Gerrit Trigger 配置，如图 4-21 所示。

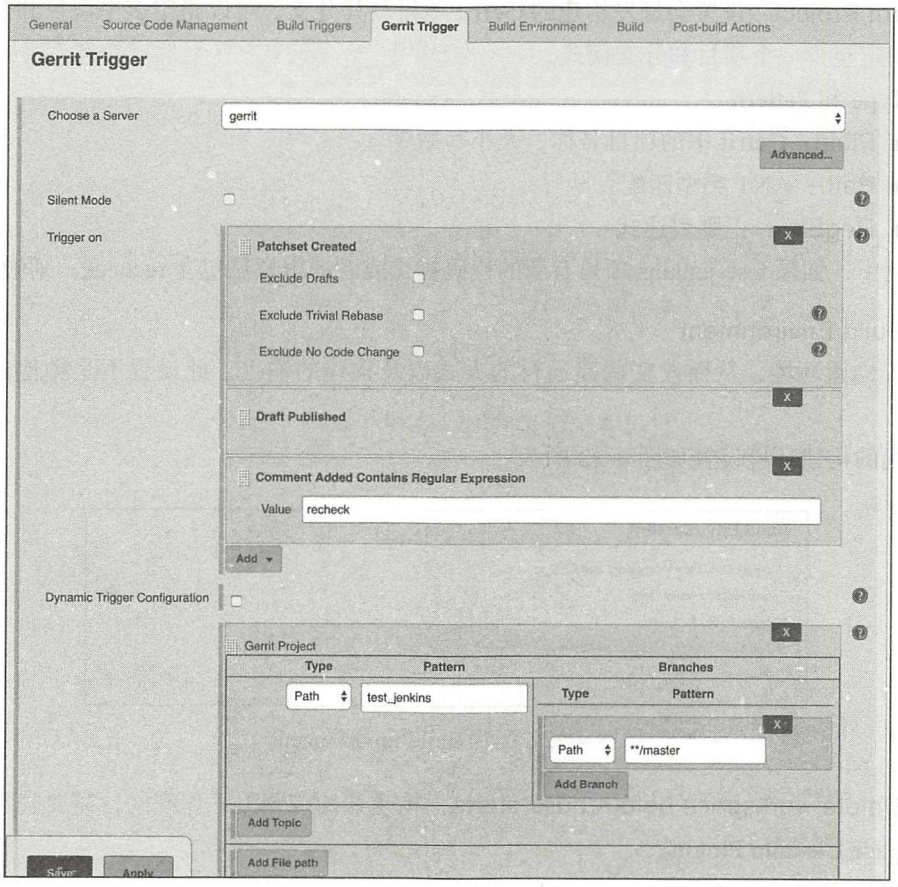


图 4-21　配置 Gerrit trigger

主要配置项说明如下：

- Choose a Server：选择 Gerrit 服务，Gerrit 服务的配置方法见 4.1.4 节。
- Trigger on：设置 Gerrit 触发事件，常用的 Gerrit 事件如表 4-2 所示。

表 4-2 常用的 Gerrit Event

Trigger on	描述
Change Abandoned	变更放弃
Change Merged	变更合入
Change Restored	放弃的变更恢复
Comment Added Contains Regular Expression	评审意见包括特定表达式
Draft Published	草稿发布
Patchset Created	变更提交
Ref Updated	引用更新
Topic changed	主题变更

Gerrit Project：设置会触发构建的 Gerrit 项目及其分支，可以指定多个项目和分支，但是必须指定至少一个项目和分支模式。

- Type 和 Pattern
  - Plain：Gerrit 中的项目名称，大小写敏感。
  - Path：ANT 类型匹配[⊖]。
  - RegExp：正则表达式。

本例中，如果 test_jenkins 项目有新的代码提交或者评审意见包含 recheck，则触发构建。

4. Build Environment

设置构建环境，从而改变构建运行的方式以及构建的输出，此设置不是排他的，可以设置多个。

常用的构建环境设置如图 4-22 所示。

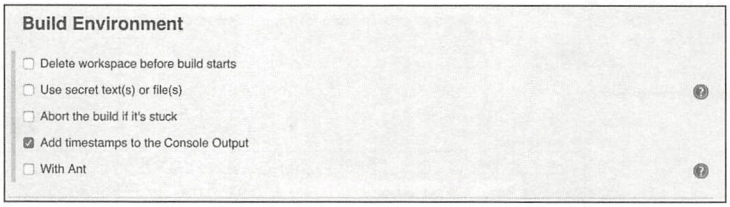


图 4-22 配置 Build Environment

- Delete workspace before build starts：构建开始前删除工作空间，需要安装 Workspace Cleanup Plugin。
- Use secret text（s）or ftle（s）：绑定安全认证码。提供 SSH 私钥、安全文件及用户名密码等方式，需要安装 Credentials Binding Plugin。
- Abort the build if it's stuck：如果构建被阻塞则中止构建，需要安装 Build timeout Plugin。
- Add timestamps to the Console Output：设置构建日志加上时间标签，需要安装

⊖ https://ant.apache.org/manual/dirtasks.html。

Timestamper Plugin。

- With Ant：为构建环境安装并配置 Apache Ant。

5. Build

设置构建方式，构建方式可以设置多个，Jenkins 会根据设置顺序执行。常用的构建方式如下所示：

- Execute Windows batch command：使用 Windows 的 batch 作为构建系统。构建执行体可是脚本文件，也可以是一段 batch 命令，如果构建执行完成后 %ERRORLEVEL% 不为 0，则被视为构建失败。
- Execute shell：Shell 命令或者脚本构建，如果脚本文件没有指明 Sha-Bang[⊖]如 #!/bin/sh-，则使用系统级的 Shell 配置。同时也可以通过 Sha-Bang 指定用其他语言运行脚本，如 #!/usr/bin/env python 表明使用 Python 运行此脚本。返回值非 0 表示构建失败。
- Invoke Ant：使用 Ant 作为构建系统，返回值非 0 表示构建失败，需要安装 Ant Plugin。
- Invoke Gradle script：使用 Gradle 作为构建系统，返回值非 0 表示构建失败，需要安装 Gradle Plugin。
- Invoke top-level Maven targets：使用 Maven 作为构建系统，返回值非 0 表示构建失败，需要安装 Artifactory Plugin。
- Run with timeout：规定时长内没有完成构建，则结束构建，需要安装 Build Timeout Plugin。本例选择 Execute shell 并通过脚本方式进行构建，如图 4-23 所示。

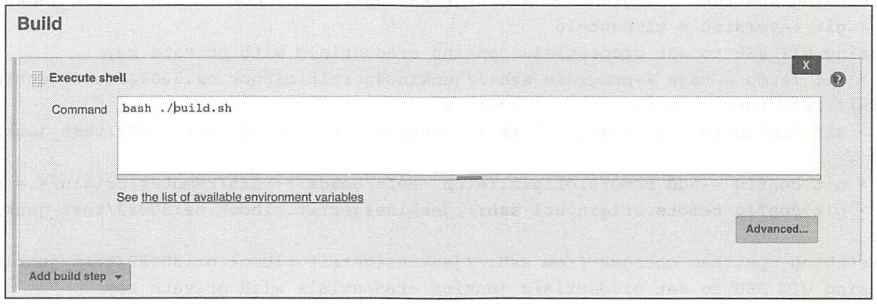


图 4-23　配置 Build

build.sh 脚本的内容如下：

```
echo "Trigger the $PROJECT job!"
```

6. Post-build Actions

定义构建完成后的动作，构建完成之后可执行的动作有很多，如发送邮件通知、日志转存、发布单元测试结果报告、触发下游项目启动构建及删除工作空间等，此阶段也可以设置多个。本例中仅设置构建失败就发送邮件通知，如图 4-24 所示。

⊖ https://en.wikipedia.org/wiki/Shebang_（Unix）。

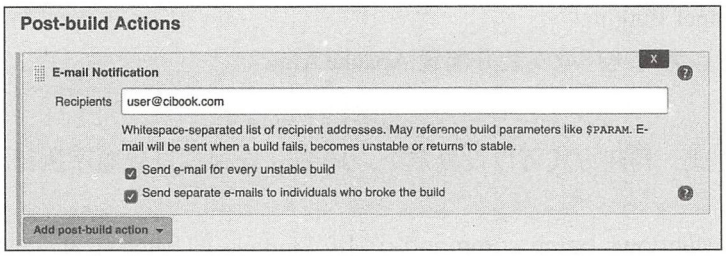


图 4-24 配置 Post-build Actions

7. 构建演示

上述配置完成后，点击按钮 Apply → Save，Job 创建成功，每当 test_jenkins 项目有新的代码变更提交或者评审中包含 recheck，就会触发新的构建，构建日志如图 4-25 所示。

同时在 Gerrit 上会出现如图 4-26 所示的评审过程记录：

Console Output

```
06:25:56 Triggered by Gerrit: http://gerrit-server-66f66896b6-xldjj:8080/7
06:25:56 Building remotely on test_label (ubuntu remote) in workspace
/home/serena/label/workspace/test_freestyle
06:25:57 Wiping out workspace first.
06:25:57 Cloning the remote Git repository
06:25:57 Cloning repository ssh://jenkins@gerrit.cibook.oz:30082/test_jenkins.git
06:25:57  > git init /home/serena/label/workspace/test_freestyle # timeout=10
06:25:57 Fetching upstream changes from ssh://jenkins@gerrit.cibook.oz:30082/test_jenkins.git
06:25:57  > git --version # timeout=10
06:25:57 using GIT_SSH to set credentials jenkins credentials with private key
06:25:57  > git fetch --tags --progress ssh://jenkins@gerrit.cibook.oz:30082/test_jenkins.git
+refs/heads/*:refs/remotes/origin/* # timeout=15
06:25:57  > git config remote.origin.url ssh://jenkins@gerrit.cibook.oz:30082/test_jenkins.git #
timeout=10
06:25:57  > git config --add remote.origin.fetch +refs/heads/*:refs/remotes/origin/* # timeout=10
06:25:57  > git config remote.origin.url ssh://jenkins@gerrit.cibook.oz:30082/test_jenkins.git #
timeout=10
06:25:57 Fetching upstream changes from ssh://jenkins@gerrit.cibook.oz:30082/test_jenkins.git
06:25:57 using GIT_SSH to set credentials jenkins credentials with private key
06:25:57  > git fetch --tags --progress ssh://jenkins@gerrit.cibook.oz:30082/test_jenkins.git
refs/changes/07/7/35 # timeout=15
06:25:57  > git rev-parse 9b70954c98d81e0244b750c86c1695f6d61cb621^{commit} # timeout=10
06:25:57 Checking out Revision 9b70954c98d81e0244b750c86c1695f6d61cb621 (refs/changes/07/7/35)
06:25:57  > git config core.sparsecheckout # timeout=10
06:25:57  > git checkout -f 9b70954c98d81e0244b750c86c1695f6d61cb621 # timeout=15
06:25:57 Commit message: "trigger build.sh"
06:25:57  > git rev-parse FETCH_HEAD^{commit} # timeout=10
06:25:58  > git rev-list --no-walk 017d6880d224c90ef20698d0d0eb8c28be2150dc # timeout=10
06:25:58 [test_freestyle] $ /bin/sh -xe /tmp/jenkins6643313612949672629.sh
06:25:58 + bash ./build.sh
06:25:58 Trigger the test_jenkins job!
06:25:58 Finished: SUCCESS
```

图 4-25 构建日志

Administrator	Uploaded patch set 35.	May 10 2:25 PM
jenkins	Patch Set 35: Build Started http://localhost:8080/job/test_freestyle_macro/6/ (1/3)	May 10 2:25 PM
jenkins	Patch Set 35: Build Started http://localhost:8080/job/test_freestyle_jjb/11/ (2/3)	May 10 2:25 PM
jenkins	Patch Set 35: Build Started http://localhost:8080/job/test_freestyle/74/ (3/3)	May 10 2:25 PM
jenkins	Patch Set 35: Verified+1 Build Successful http://localhost:8080/job/test_freestyle_jjb/11/ : SUCCESS http://localhost:8080/job/test_freestyle_macro/6/ : ...	May 10 2:25 PM
Administrator	Patch Set 34: recheck	May 10 2:26 PM
jenkins	Patch Set 34: -Verified Build Started http://localhost:8080/job/test_freestyle_jjb/12/ (1/3)	May 10 2:26 PM
jenkins	Patch Set 34: Build Started http://localhost:8080/job/test_freestyle_macro/7/ (2/3)	May 10 2:27 PM
jenkins	Patch Set 34: Build Started http://localhost:8080/job/test_freestyle/75/ (3/3)	May 10 2:27 PM
jenkins	Patch Set 34: Verified+1 Build Successful http://localhost:8080/job/test_freestyle_jjb/12/ : SUCCESS http://localhost:8080/job/test_freestyle_macro/7/ : ...	May 10 2:27 PM
jenkins	Patch Set 34: -Verified Build Started http://localhost:8080/job/test_freestyle/76/ (3/3)	May 10 2:27 PM
jenkins	Patch Set 34: Verified+1 Build Successful http://localhost:8080/job/test_freestyle_jjb/12/ : SUCCESS http://localhost:8080/job/test_freestyle_macro/7/ : ...	May 10 2:27 PM

图 4-26　Gerrit 构建记录

从图 4-26 中可以看出，每次有新的 PatchSet 提交或者评审中包含 recheck 都会触发 test_freestyle 构建，构建过程中 Jenkins 会发送两条消息给 Gerrit，一条表示构建开始，即评审中包含 Build Started 那条，第二条表示构建结束并进行投票，即评审中包含 Verified + 1 那条，如果构建失败则会返回 Verified-1 信息。

一个简单的 Job 创建、配置、构建过程就是如此。当然，真实的使用场景（如 OpenStack 由于涉及项目很多、触发条件很多、且差异性很大等）情况要复杂得多，而且手工创建有不方便移植、不方便管理、不可追溯和配置不可复用等问题，为了解决这些问题 OpenStack Infra 团队开发了组件 Jenkins-job-builder 来对这些 Job 进行代码化自动化管理，我们将在下一章介绍。

4.2　Jenkins Job Builder

Jenkins 创建的 Job 以 XML 文件的格式保存在 $JENKINS_HOME/jobs/{job_name}/config.xml 中。XML 文件不易于阅读和维护，因此 Jenkins Job Builder（简称 JJB）采用易于阅读的 YAML 或者 JSON 格式来配置 Job，然后转化为 Jenkins 易于理解的 XML 文件，更新到 Jenkins 中。YAML 和 JSON 都可以用来定义 JJB Job，且它们之间是可以相互转换的，由于 YAML 使用更加广泛，所以本书只介绍 YAML 文件的使用方式。

4.2.1　安装 JJB

JJB 是使用 Python 开发的开源软件，和其他所有 Python 包一样都支持 PIP 开发包安装和源码安装。PIP 安装如下所示：

```
pip install jenkins-job-builder
```

安装成功之后，就能够使用 jenkins-jobs 这个命令来测试 YAML 文件或者创建更新 Job 了。

4.2.2　配置 JJB

JJB 默认会按顺序从以下代码所描述的位置找配置文件：

```
1. ~/.config/jenkins_jobs/jenkins_jobs.ini
2. <script directory>/jenkins_jobs.ini
3. /etc/jenkins_jobs/jenkins_jobs.ini
```

当然也支持自定义路径，在命令行中使用 --conf jenkins_jobs.ini 参数即可。配置文件中指定 Jenkins 的 URL 地址和登录信息，从而 JJB 可以将 Job 更新到指定的 Jenkins。简单的配置信息如代码清单 4-3 所示。

代码清单 4-3　jenkins_jobs.ini

```
[job_builder]
ignore_cache=True
recursive=False

[jenkins]
user=jenkins
password=jenkins
url=http://jenkins.cibook.oz:8080/jenkins
```

4.2.3　使用 JJB

定义 Job 的 YAML 文件编写完成之后，进行 Job 的上传之前，可以通过 jenkins-jobs 命令来验证配置格式和内容是否正确，验证成功之后就可以把 Job 创建或者更新到 Jenkins 上了。

1. Job 定义验证

Job 定义通过 test 命令来进行验证，如下面代码所示：

```
jenkins-jobs test yaml_path/foo.yaml # 验证 Job 定义，并将结果打印到控制台
jenkins-jobs test yaml_path/foo.yaml -o xml_path # 验证 Job 定义，并将结果保存到目
                                                 录 xml_path
```

2. 创建和更新 Job

update 命令用来创建或者更新 Job，如下面代码所示：

```
jenkins-jobs update yaml_path/foo.yaml # 创建或更新 yaml_path 目录下 foo Job
jenkins-jobs update yaml_path/         # 创建或更新 yaml_path 目录下所有 Job
jenkins-jobs update yaml_path/ foo bar # 创建或更新 yaml_path 目录下 foo 和 bar Job
```

3. 删除 Job

Job 的删除通过 delete 或者 delete-all 命令进行，如下面代码所示：

```
jenkins-jobs delete foo # 删除 foo Job
jenkins-jobs delete-all # 删除 Jenkins 上所有 Job
```

4.2.4　JJB 语法详解

JJB 通过一系列自定义的语义来描述 Job，可以在一个 YAML 文件里完成所有 Project

非卖品，仅供非商业用途或交流学习使用

或者 Job 定义，也可以是多个 YAML 文件来实现 JJB Job 管理。在进行 XML 文件转换时，可以指定单个 YAML 文件，也可选择存放目录。

1. Job

在 JJB 中，Job 通过一些通用参数和模块进行描述，Job 定义的模块有很多，在此不一一介绍，有需要的读者可自行查阅官方文档[⊖]，其常用的模块及执行顺序如下：

1）Parameters/Properties

2）SCM

3）Triggers

4）Wrappers

5）Builders

6）Publishers

手动创建的 test_freestyle 的完整 Job 定义如代码清单 4-4 所示。

代码清单 4-4　test_jenkins.yaml

```
- job:
    name: test_freestyle
    project_type: freestyle
    disabled: false
    node: ubuntu
    parameters:
        - string:
            name: PROJECT default: test_jenkins
            description:
            project name in scm repo
    properties:
        - build-discarder:
            days-to-keep: 30
            num-to-keep: 100
    scm:
        - git:
            credentials-id: 'jenkins-ci'
            url: 'ssh://jenkins@gerrit.cibook.oz:30082/test_jenkins.git'
            branches:
                - '**/master'
                choosing-strategy: 'gerrit'
                refspec: '$GERRIT_REFSPEC'
                skip-tag: 'true'
                wipe-workspace: 'true'
    triggers:
        - gerrit:
            server-name: 'gerrit'
            trigger-on:
```

⊖ https://docs.openstack.org/infra/jenkins-job-builder/。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
                - patchset-created-event:
                        exclude-drafts: 'false'
                        exclude-trivial-rebase: 'false'
                        exclude-no-code-change: 'false'
                - draft-published-event
                - comment-added-contains-event:
                    comment-contains-value: 'recheck'
            projects:
                - project-compare-type: 'ANT'
                    project-pattern: 'test_jenkins'
                    branches:
                        - branch-compare-type: 'ANT'
                        branch-pattern: '**/master'
    wrappers:
        - timestamps
    builders:
        - shell: 'bash build.sh'
    publishers:
        - email:
            recipients: user@cibook.com
            notify-every-unstable-build: True
            send-to-individuals: true
```

下面就代码清单 4-4 的各个模块分别进行介绍。

2. 通用参数

一组简单的字符串或者布尔值的键值对，对应 4.1.7 节中的部分配置如下：

- name：Job 名称，对应 Job 配置页面上定义的 Project name。
- project-type：Job 类型，默认为 freestyle 类型，对应手动创建 Job 时选择的 Job 类型。
- disabled：布尔值，如果设置为 true，则此 Job 不会被触发。默认为 false，对应 Job 配置页面上定义的 Disable this project。
- concurrent：布尔值，设置此 Job 的多个构建是否可以并行执行，默认为 false，对应 Job 配置页面上定义的 Execute concurrent builds if necessary。
- node：通过 Slave 名称或者标签，绑定 Job 运行在哪些 Slave 上，对应 Job 配置页面上定义的 Restrict where this project can be run。

上面手动创建的 test_freestyle Job 中通用参数部分的描述如代码清单 4-5 所示。

代码清单 4-5　general_parameters.yaml

```
- job:
    name: test_freestyle
    project_type: freestyle
    disabled: false
    node: ubuntu
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

3. Parameters

JJB 通过 Parameters 对应第 4.1.7 小节中 This project is parameterized 的参数定义，如表 4-3 所示。

表 4-3　JJB 中常用的 Parameters

Parameters	功能	Job 配置
bool	定义布尔变量（true/false）	Bool Parameter
choice	定义枚举变量	Choice Parameter
credentials	选择 credentials	Credentials Parameter
file	指定上传文件路径	File Parameter
label	设置执行构建的 Slave 标签	Label
node	列出可执行构建的所有 Slave	Node
string	定义字符串变量	String Parameter

如在 test_freestyle 中定义的参数 $PROJECT，在 JJB 中可以进行如代码清单 4-6 所示的描述。

代码清单 4-6　project_parameter.yaml

```
parameters:
    - string:
        name: PROJECT
        default: test_jenkins
        description: "project name in scm repo"
```

再如，绑定 test_freestyle 到安装 Ubuntu 系统的 Slave 上执行也可以通过定义 label 参数实现，如代码清单 4-7 所示。

代码清单 4-7　label_parameter.yaml

```
parameters:
    - label:
        name: test_label
        default: ubuntu
        description: "Run this job on test_label Slave."
```

如果 Job 中同时设定了 label 参数和上一节中的 node 参数，则 label 起决定作用。

4. Properties

JJB 通过 Properties 描述 Job 的属性配置，对应 4.1.7 节中属性配置部分的 JJB 描述如表 4-4 所示。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

表 4-4 JJB 中常用的 Properties

Proper-ties	功能	Job 配置
build-blocker	控制 Job 顺序执行	Block build when upstream project is building Block build when downstream project is building
build-discarder	构建日志保持时长配置	Discard old builds
github	项目在 GitHub 上的 URL	GitHub project
throttle	可执行的并行构建数目	Throttle Concurrent Builds
zeromq-event	通过 ZMQ 发布 Job 事件	ZMQ Event Publisher

test_freestyle 中构建日志保存时长的定义在 JJB 中的描述如代码清单 4-8 所示。

代码清单 4-8 properties.yaml

```
properties:
    - build-discarder:
        days-to-keep: 30
        num-to-keep: 100
```

5. SCM

SCM 模块对应 Job 配置中的 Source Code Management，正如 4.1.7 节中讲到的，Jenkins 支持的 SCM 很多，每个配置都不一样，在此不一一说明，此处仅根据 test_freestyle 的 SCM 配置来简单说明一下 JJB 的 SCM 描述方法。图 4-20 配置的 JJB 描述如代码清单 4-9 所示。

代码清单 4-9 gerrit_scm.yaml

```
scm:
    - git:
        credentials-id: jenkins-ci
        url: ssh://jenkins@gerrit.cibook.oz:30082/test_jenkins.git
        branches:
            - '**/master'
        choosing-strategy: 'gerrit'
        refspec: '$GERRIT_REFSPEC'
        wipe-workspace: 'true'
```

代码清单 4-9 中的参数说明如下：

- git：指明代码库是 Git 类型。
- credentials-id：在 4.1.4 节中创建的访问代码库的 Credentials，对应 Job 配置中的 Creden-tials。
- url：代码库的访问链接，对应 Job 配置中的 Repository URL。
- branches：待构建的分支，对应 Job 配置中的 Branches to build。
- choosing-strategy：配置 Gerrit 作为 SCM 时必须配置，对应 Job 配置中的 Choosing

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

strategy，gerrit 对应选择项 Gerrit Trigger。

- refspec：本地引用和远端引用的映射，对应 Job 配置中的 Refspec。
- wipe-workspace：构建开始前删除工作空间，对应 Job 配置中的 Wipe out repository & force clone。

6. Triggers

Triggers 对应 4.1.7 节，常用的如表 4-5所示。

表 4-5　JJB 中常用的 Triggers

Triggers	功能	Job 配置
timed	周期触发	Build periodically
pollscm	周期轮询 SCM 是否变更	Poll SCM
build-result	其他 Job 构建结束	Build after other projects are built
gerrit	Gerrit 事件	Gerrit event
github	github 库代码更新	GitHub hook trigger for GITScm polling
gitlab	gitlab 库代码更新	Build when a change is pushed to GitLab

test_freestyle 中的 Gerrit 库代码变更触发的定义如代码清单 4-10 所示。

代码清单 4-10　gerrit_trigger.yaml

```
triggers:
    - gerrit:
        server-name: 'gerrit'
        trigger-on:
            - patchset-created-event:
                    exclude-drafts: 'false'
                    exclude-trivial-rebase: 'false'
                    exclude-no-code-change: 'false'
            - draft-published-event
            - comment-added-contains-event:
                    comment-contains-value: 'recheck'
        projects:
            - project-compare-type: 'ANT'
              project-pattern: 'test_jenkins'
              branches:
                  - branch-compare-type: 'ANT'
                    branch-pattern: '**/master'
```

代码清单 4-10 参数说明如下：

- server-name：Gerrit 服务的名字，对应 Job 配置 Choose a Server。
- trigger-on：设置 Gerrit 触发事件，对应 Job 配置 Trigger on，表 4-2 中的 Gerrit 触发事件在 JJB 中的定义如表 4-6 所示。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

表 4-6 JJB 中常用的 Gerrit Events

Trigger on	描述	Job 配置
change-abandoned-event	变更放弃	Change Abandoned
change-merged-event	变更合入	Change Merged
change-restored-event	放弃的变更恢复	Change Restored
comment-added-contains-event	评审意见包括特定表达式	Comment Added Contains ...
draft-published-event	草稿发布	Draft Published
patchset-created-event	变更提交	Patchset Created
ref-updated-event	引用更新	Ref Updated

- projects：Gerrit 项目和分支描述，对应于 Job 配置的 Gerrit Project，其匹配类型和 Job 配置的对应关系如表 4-7 所示。

表 4-7 Patterns 对应关系

JJB 描述	Job 配置	匹配规则
ANT	Path	Apache ANT 匹配
PLAIN	Plain	项目名完全匹配
REG_EXP	RegExp	正则表达式匹配

7. Wrappers

对应 4.1.7 节，常用 Wrappers 的 JJB 描述如表 4-8 所示。

表 4-8 JJB 中常用的 Wrappers

Wrappers	功能	Job 配置
credentials-binding	绑定 Credentials	Use secret text（s）or file（s）
timeout	设置构建超时时长	Abort the build if it’s stuck
buildstamps	设置打印日志加时间戳	Add timestamps to the Console Output
workspace-cleanup	构建前清理工作空间	Delete workspace before build starts

为 test_freestyle 的输出日志加时间信息的描述如代码清单 4-11 所示。

代码清单 4-11 timestamps_wrapper.yaml

```
wrappers:
    - timestamps:
```

8. Builders

对应 4.1.7 节，常用 Builders 的 JJB 描述如表 4-9 所示。

表 4-9 JJB 中常用的 Builders

Builders	功能	Job 配置
batch	执行 Python 命令或脚本	Execute Windows batch command

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

（续）

Builders	功能	Job 配置
shell	执行 Shell 命令或脚本	Execute shell
ant	执行 ANT 任务	Invoke Ant
gradle	执行 Gradle 任务	Invoke Gradle script
maven-target	执行 top-level Maven 任务	Invoke top-level Maven targets

test_freestyle 的构建执行如代码清单 4-12 所示。

代码清单 4-12　shell_builder.yaml

```
builders:
    - shell: 'bash build.sh'
```

一次构建执行可以触发多个动作，这些动作可以不相同。如代码清单 4-13 所示，一次 Job 触发了两个动作，一个编译 docker 容器，一个打印任务执行完成。

代码清单 4-13　multi_buiders.yaml

```
builders:
    -     docker-build-publish:
        repo-name: 'test'
        repo-tag: 'test-tag'
        no-cache: true
        no-force-pull: false
        skip-build: false
        skip-decorate: false
        skip-latest: false
        skip-tag: false
        file-path: '/tmp/'
        build-context: '/tmp/'
    -     shell: |
        'echo docker build finish'
```

9. Publishers

Publishers 定义了 Job 构建完成后的发布工作，常用的 Publishers 如表 4-10 所示。

表 4-10　JJB 中常用的 Publishers

Builders	功能	Job 配置
cobertura	产生代码覆盖率报告	Publish Cobertura Coverage Report
email	构建失败通知	E-mail Notification
github-notifier	通知 GitHub 构建结果	Set GitHub commit status
junit	发布 JUnit 测试结果	Publish JUnit test result report
postbuildscript	定义构建完成后的脚本操作	Post Build task
scp	通过 SCP 上传文件	Publish artifacts to SCP Repository
workspace-cleanup	构建完成后清理工作空间	Delete workspace when build is done

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

构建失败则发送邮件通知的定义如代码清单 4-14 所示。

代码清单 4-14 email_publish.yaml

```
publishers:
    - email:
        recipients: user@cibook.com
        notify-every-unstable-build: True
        send-to-individuals: true
```

10. Job Template

如果一个项目中多个 Job 功能相同，只是名称、版本、分支等有区别，或者多个项目需要执行相同或类似的构建 Job，这样的情况下就要考虑使用 Job Template 来定义 Job，在定义 Project 的时候通过变量置换来实例化 Job，以增加代码的重用性，减少维护的工作量。Job Template 的语义和 Job 的语义相同，只是在定义中增加了变量。例如，如果多个项目都通过 Shell 方式构建，且其中 test_jenkins 项目有 master 和 stable 两个分支，且两个分支都需要进行代码变更构建检查，可通过代码清单 4-15 的内容来进行描述。

代码清单 4-15 job_template.yaml

```
- job-template:
    name: 'test_freestyle-{pname}-{stream}'
    builders:
        - shell: 'bash build.sh'
```

在这种描述下，{pname} 和 {stream} 参数需要通过 Project 的参数定义进行实例化，Project 的定义请参见 4.2.4 节。

11. Job Group

如果一组 Job 总是在一起使用，则可以定义一个 Job Group。如 OpenStack 社区大部分项目都是用 Python 编码和 RST 写文档，pylint 检查和 RST 格式检查就可以组织为一个 Job Group，如代码清单 4-16 所示。

代码清单 4-16 job_group.yaml

```
- job-group:
    name: '{pname}-lint-check'
    jobs:
        - '{pname}-pylint-check'
        - '{pname}-rst-check'
```

12. Project

Project 的使用是为了把相关的 Job 集中在一起，并为 Job Template 提供实例化参数，如代码清单 4-15中的 Job Template 可以通过代码清单 4-17 所示的 Project 描述进行实例化。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

代码清单 4-17　project.yaml

```
- project:
    name: test_jenkins_project
    pname: test_jenkins
    stream:
        - master
        - stable
    jobs:
        - 'test-freestyle-{pname}-{stream}'
```

在这种描述下，JJB 解析的时候，{pname} 参数实例化为 test_jenkins，{stream} 参数通过 master 和 stable 分别进行两次实例化，因此在 Jenkins 中会增加 test-freestyle-test_jenkins-master 和 test-freestyle-test_jenkins-stable 两个 Job。

13. Macros

JJB 通过宏扩展的方式管理复杂或重复的 Job 动作，宏定义的格式分为定义宏应用的模块，定义宏名和定义宏定义体三部分，例如 test_freestyle 中参数 PROJECT 的宏定义如代码清单 4-18 所示。

代码清单 4-18　macro_parameter.yaml

```
# 定义宏应用模块，指明此处是  Parameters 模块宏定义
- parameter:


    # 定义宏定义名称为  project-parameters
    name: project-parameters


    # 定义宏定义体
    parameters:
        - string:
            name: PROJECT
            default: test_jenkins
            description: Define job type
```

宏定义完成之后，就可以在 Job 中进行引用了，如代码清单 4-19 所示。

代码清单 4-19　macro_parameter_reference.yaml

```
- job:
    name: test_freestyle
    # 宏引用，引用宏定义 project-parameters
    parameters:
        - project-parameters:
```

在 JJB 解析的时候，宏引用会被替换为宏定义体，如代码清单 4-20 所示。

仅供非商业用途或交流学习使用

代码清单 4-20 macro_extension.yaml

```
- job:
    name: test_freestyle
    # 宏扩展部分，以下是通过 project-parameters 的宏定义体进行的宏扩展，
    # 由此可以看出它是宏定义体的纯粹的复制和粘贴
    parameters:
        - string:
        name: PROJECT
        default: test_jenkins
        description: Define job type
```

通过以上描述，使用宏定义和 Project 定义重构 test_freestyle 中的核心 Parameter、SCM 和 Triggers 部分，使其适用于不同的项目，如代码清单 4-21 所示。

代码清单 4-21 macros.yaml

```
- parameter:
    name: project-parameters
    parameters:
        - string:
            name: PROJECT
            default: '{project}'
            description: Define job type

    -      scm:
    name: gerrit-scm scm:
        - git:
            credentials-id: jenkins-ci
            url: 'ssh://jenkins@gerrit.cibook.oz:30082/$PROJECT.git'
            branches:
                - '**/master'
            choosing-strategy: 'gerrit'
            refspec: '$GERRIT_REFSPEC'
            skip-tag: 'true'
            wipe-workspace: 'true'


-  trigger:
    name: patchset_created_trigger
    triggers:
        - gerrit:
            server-name: 'gerrit'
            trigger-on:
                - patchset-created-event:
                        exclude-drafts: 'false'
                        exclude-trivial-rebase:'false'
                        exclude-no-code-change: 'false'
                - draft-published-event
                - comment-added-contains-event:
                        comment-contains-value: 'recheck'
```

```
        projects:
                - project-compare-type: 'ANT'
                    project-pattern: '{project}'
                    branches:
                        - branch-compare-type: 'ANT'
                            branch-pattern: '**/master'
```

其引用方式如代码清单 4-22 所示。

代码清单 4-22　test_jenkins_macro.yaml

```
- job-template:
    name: 'test_freestyle-{pname}'
    project_type: freestyle
    disabled: false
    parameters:
        - project-parameters:
            project: '{pname}'
    scm:
        - gerrit-scm
    triggers:
        - patchset_created_trigger:
            project: '{pname}'
- project:
    name: test_jenkins
    pname: test_jenkins
    jobs:
        - 'test_freestyle-{pname}'
```

14. Defaults

集中收集 Job 的一些默认属性和动作，在创建 Job 时提供这些值，当然，在 Job 定义中也可以重置这些默认属性。如果使用 global 指定了一组默认值，则在所有 Job（和 Job Template）中都可以使用，除非在 Job 中进行了重置。简单的 Defaults 的例子如代码清单 4-23 所示。

代码清单 4-23　defaults.yaml

```
- defaults:
    name: global
    arch: 'i386'
- job-template:
    name: 'build-{arch}'
    builders:
        - shell: "echo Build arch {arch}."
- project:
    name: project1
    jobs:
        - 'build-{arch}'
```

```
- project:
    name: project2
    jobs:
        - 'build-{arch}':
            arch: 'amd64'
```

通过上面的 JJB 描述，project1 中的 arch 使用全局默认值，因此将运行 build-i386，而 project2 由于对 arch 进行了重置，运行 build-amd64。

4.3 Python Jenkins

上文我们提到过 Nodepool 是通过 HTTP 给 Jenkins 发送消息的，OpenStack 社区开发了 python-jenkins 组件来实现这些 HTTP 消息的发送和接收。

4.3.1 安装 python-jenkins

python-jenkins 支持 PIP 开发包安装和源码安装。PIP 安装如下所示：

```
pip install python-jenkins
```

4.3.2 使用 python-jenkins

python-jenkins 库提供了一种使用 RESTful API 管理 Jenkins 服务的方法，它提供的服务主要包括：

- 版本管理：获取 Jenkins 运行版本。
- Job 管理：Job 管理主要包括创建、配置、更新、删除 Job，获取 Job 的构建状态、构建日志和构建结果，启动或者取消 Job 构建等。
- 插件管理：主要是安装或者卸载插件。
- Slave 管理：创建、删除或者配置 Slave。

python-jenkins 的调用方式大致相同，下面我们仅以获取 Jenkins 的运行版本为例（如代码清单 4-24）说明其调用方式，详细的接口说明请参见官方文档[⊖]。

代码清单 4-24 python-jenkins 应用举例

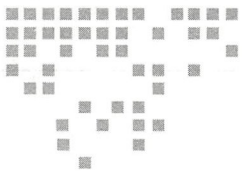
```
import jenkins


server = jenkins.Jenkins(url='http://jenkins.cibook.oz:8080',
                         username='user',
                         password='pass',
```

⊖ http://python-jenkins.readthedocs.io/en/latest/。

```
                              timeout=10)     # 实例化 Jenkins 通信句柄
version = server.get_version()                # 获取 Jenkins 服务的运行版本
print('Hello Jenkins {}'.format(version))
```

4.4　本章小结

本章通过 Gerrit + Jenkins 的方式对典型的持续集成系统进行了系统介绍，并通过一个简单但是经典的 test_freestyle Job 对 Jenkins 上 Job 的配置及配套组件 JJB 的使用做了较为全面的介绍。



Chapter 5 第 5 章

门控系统（Zuul）

本章将详细说明 Zuul[⊖]组件，它是 OpenStack 社区的一个基础设施项目，致力于实现项目及关联项目之间的守护者，仅当与变更相关的所有测试都通过后，它才会将变更合入到相应的目标分支。实现这种机制的系统称之为门控（gating）系统。

本章涉及大量的 Git 和 Gerrit 系统的术语和操作，阅读到本章节时默认您已经阅读了第 3 章的内容，如果您不熟悉，可以在任何时候回到第 3 章进行了解。

目前，大部分软件开发项目都是由开发人员负责把代码合入到主线分支，然后对这个变更进行回归测试，如果引入了故障，再通过一系列的补丁来修复。在此过程中，回归测试可以通过 CI 来自动化运行。一旦主线分支被破坏，对开发人员而言将是一件非常令人沮丧的事，这将导致其他的提交无法得到及时验证，从而降低整个团队的效率，在开发人员比较多或者提交频率比较高的项目中尤其明显。

为此，许多项目可能要求开发人员在合入代码前进行测试以保证代码不产生恶化，这是一种非正式的、通过人为来保证的门控系统，随着更多的开发人员加入，变更越来越频繁，测试任务越来越复杂，这种方式水平扩展能力受限，因而无法保证开发人员的开发效率。Zuul 的门控系统机制正是着眼于帮助项目和开发人员解决该困境。采用 Zuul，就可以对这些测试工作进行自动化编排，并确保即使有大量的变更也能得到正确的并行处理。

Zuul 保证主线分支对所有开发人员开放，所有工作都可以基于主线进行开发。一个变更只有通过所有回归测试才会被合并到主线，避免主线由于合入补丁而可能产生的恶化。

你可能会有疑问，使用 Gerrit + Jenkins + 守护任务的组合，是不是也能完成 Zuul 的这种门控系统的功能，通过本章的学习，你能了解这其中的差别。

⊖ https://docs.openstack.org/infra/zuul/。

注解　社区最新发布的 OpenStack Zuul 文档是基于 Zuul V3 的，本章如不特别说明，Zuul 都是指 Zuul V2 版本，需要通过检出 Zuul V2 版本的代码库来进行阅读和参考，也可以直接参考[⊖]。

5.1　Zuul 组件介绍

Zuul 是由若干组件组成的一个分布式系统，如图 5-1 所示。

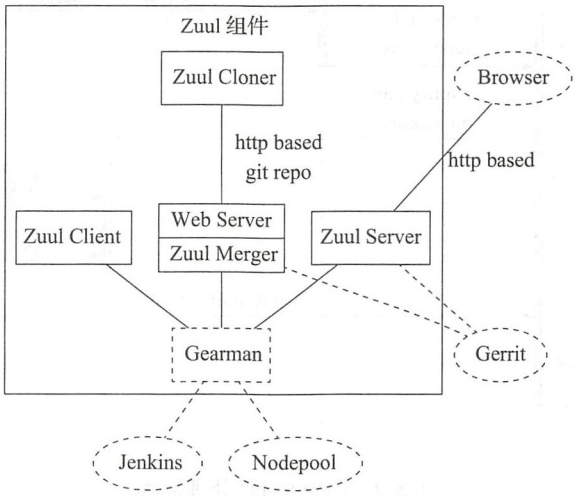


图 5-1　Zuul 组件及外部关系

Zuul 各组件的说明如下：

- Zuul Server：守护进程，与 Gerrit 和 Jenkins 等外部模块通信；负责从 Gerrit 接收事件，通知 Jenkins 启动任务，并从 Jenkins 收集结果，在任务运行结束后把结果上报到 Gerrit 服务器。
- Zuul Merger：为测试任务准备 Git 库，这要求为它提供一个 Web 服务器，以满足测试任务可基于 http 进行 Git 克隆操作。
- Zuul Cloner：这是一个命令行工具，它用于设置测试任务的工作目录（workspace），并从 Zuul Merger 准备好的 Git 库中进行克隆操作。
- Zuul Client：这是 Zuul 的简单的命令行客户端，可以用来修改 Zuul 的行为，比如：添加 / 删除队列中的元素，调整队列中元素的次序。
- Gearman：可选的内置 Gearman 服务器。

当 Zuul 收到 Gerrit 触发事件时，这些组件的协作流程如图 5-2 所示。

⊖ http://git.openstack.org/cgit/openstack-infra/zuul/tree/doc/source?h=2.6.0。

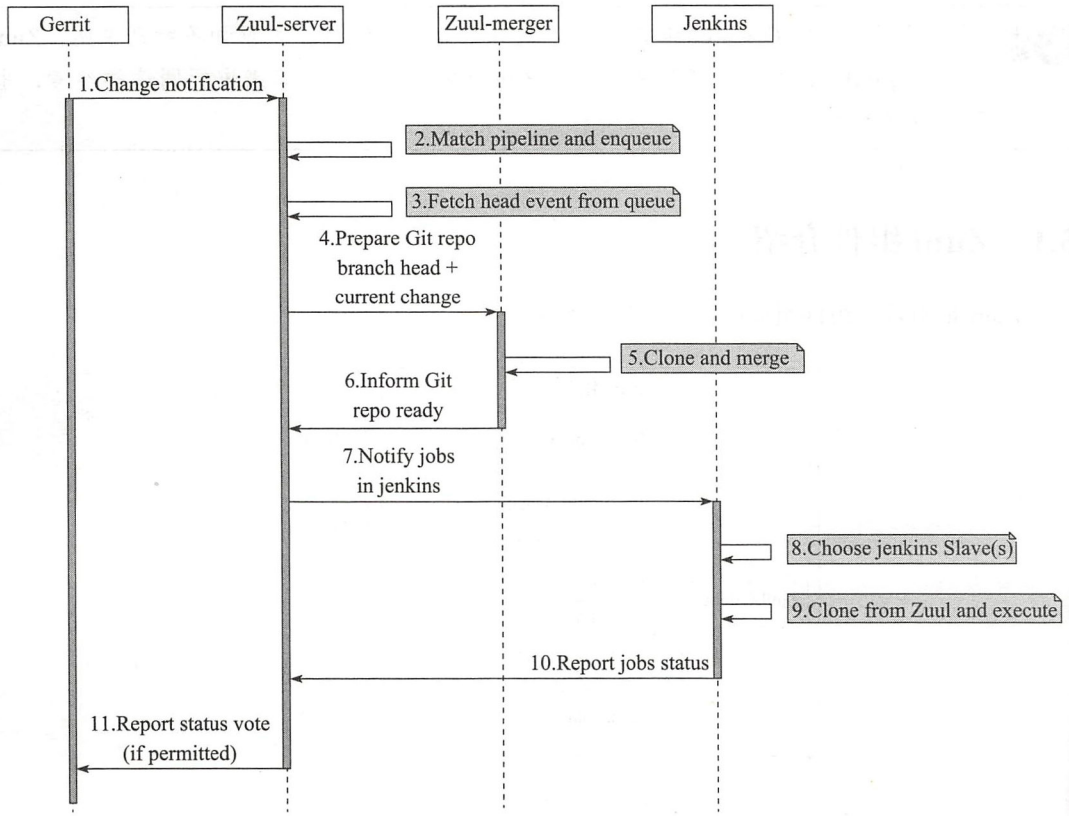


图 5-2 Gerrit 事件处理流程

图 5-2 中的流程说明如下：

1）Gerrit 向 Zuul-server 发送变更通知消息，这些消息可能包括：新增代码提交（review）、针对某个提交的补丁（patchset）、新增评论（comments）、代码合入（git merge）、提交作废（abandon）等。

2）Zuul Server 收到 Gerrit 的事件后，分析事件类型，如果是变更相关消息，则匹配是否存在与此事件相关联的队列；若存在，则把事件放在队列尾。

3）Zuul Server 从匹配的队列头取出一个事件进行处理。

4）Zuul Server 请求 Zuul Merger 为出队的事件准备 Git 代码仓库。

5）Zuul Merger 把目标分支的最新代码和本次变更的代码进行代码合并。

6）Zuul Merger 通知 Zuul Server Git 代码仓库准备就绪。

7）Zuul Server 发消息给 Jenkins 执行相关的任务。

8）Jenkins 选择一个满足条件的 Slave 执行相关的任务。

9）Jenkins 从 Zuul Merger 克隆代码并执行相关的测试服务。

10）Jenkins 在任务执行完成后向 Zuul Server 报告测试执行结果。

11）Zuul Server 把任务执行结果反馈给 Gerrit，如果 Zuul Server 具有 voting 权限，还要向 Gerrit 返回投票结果（±1/±2）。

以上是一个简化的执行流程，忽略了 Zuul 和外部模块的很多处理细节，但本流程足以描述 Gerrit 事件触发后，一个门控系统所需要完成的功能。流程中涉及的术语和过程，将在后续的章节进行描述。

5.1.1　Zuul 工作原理

Zuul 通过 pipeline 定义工作流，可以说 pipeline 是 Zuul 中最重要的概念。pipeline 定义了一个可应用于一个或多个项目（project）的工作流过程，是对执行任务的条件、验收准则和通知的一个抽象描述，以便于自动化执行。以 OpenStack 社区为例，最常用的三个 pipeline 如下：

- check pipeline：描述一个新的变更必须要通过哪些测试。
- gate pipeline：实现项目门控，当变更的所有测试都通过后，自动把代码合并（merge）到对应分支。
- post pipeline：当变更合并后进行发布处理，比如重新发布项目文档或进行部署。

check、gate 和 post 不是 Zuul 内置的关键字，可以是任意名称，这些名称只是为较准确地反映操作过程、步骤或期望，这里只是列举 Zuul pipeline 几个常见的示例。一旦定义了 pipeline，任何项目（参见 5.6.3 节）都可以使用这个 pipeline，同时指定这个项目的哪些任务（参见 5.6.2 节）在这个 pipeline 下运行。

每个 pipeline 都有自己的触发器，这些触发器描述了把某个事件加入 pipeline 的规则。举例来说，当一个补丁被提交到 Gerrit 之后，就触发了一个 patchset-created 事件。当 Zuul 收到这个事件时，一个配置了 patchset-created 事件触发器的 pipeline 就会把这个变更放到队列中。如果这个 pipeline 和 patchset 所属项目的任务关联，这些任务就会被触发。Zuul 除支持 Gerrit 触发器外，还支持定时器和 Zuul 内部事件触发，详细的列表可以参见 5.4 节。

pipeline 的队列是一种基本的数据结构，具有先进先出（FIFO，First-In-First-Out）的特点，只允许在尾部插入，头部删除，如图 5-3 所示。但需要说明的是，Zuul 可以根据需要，或通过命令行调整队列中某个元素的优先级，例如把某个事件调整到队列头部、删除某个事件等。

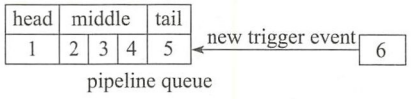


图 5-3　pipeline 队列示例

在 pipeline 触发的所有任务运行完成之后，pipeline 的报告器（参见 5.5 节）负责上报所有任务运行的结果。如果一个 pipeline 配置了 Gerrit 报告器，那么这个 pipeline 会向 Gerrit 系统提交这个变更的评审结果（review comment）。如果 Zuul 还具有 Gerrit 系统上的投票（voting）权限，还会提交投票结果（±1 和 ±2 分别表示通过 / 不通过和批准 / 不批准），具体的配置参考 3.2 节。

每个进入 pipeline 队列的条目都会与一个引用（git ref）关联，每个 ref 都指向一个特定的变更，或者只是某个分支或标签的提示。触发器事件决定 ref 代表分支、标签还是合并提交。

Zuul 在运行任务前为其准备好这个 ref。Zuul 创建的 ref 是以 Z 打头，后接一串 32 字节的 16 进制数（Git 本身的 ref 是 40 字节），为便于区分，后续称之为 zuul ref。对于一个新变更，Zuul 会把变更合并到目标分支，这就是说，所有的任务都是基于该变更合并到目标分支之后执行的。当变更被创建后，远端代码库（也就是 Gerrit 服务器中的代码库）可能已经合入了一些其他变更，从某种意义上来说，Zuul 准备的代码库可能与变更初始创建时不同。如果在 pipeline 中的条目依赖于其他条目，那么所有这些依赖变更都会被包含在 Zuul 所准备的 Git 代码库中。

图 5-4 是一个典型的 Git 修改历史示意图。提交 b 和提交 d，都是基于提交 a 进行的修改。同时主线分支（示例为 master）在提交 d 被合入前，依次合入了 b、c、e 三个提交。提交 d 创建时基于提交 a 所引用的代码库，此时若需要对提交 d 进行测试，主线分支的最新提交（tip）已经发生变化，直接使用变更 d 的代码进行测试将无法验证提交 d 与主线分支的兼容性。Zuul-merger 会获取目标分支的最新提交，即提交 e 所引用的当前状态和提交 d 的修改进行合并，得到 Git 代码库的当前状态 d′，并把 d′ 提供给测试任务使用。

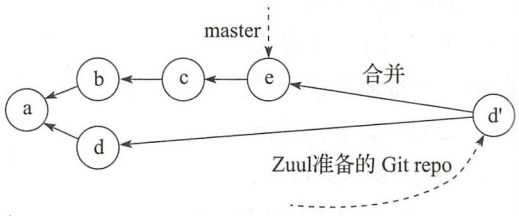


图 5-4　Zuul 代码合并处理流程

如图 5-5 所示，执行具体测试的 Jenkins 任务并不直接从 Gerrit 服务器获得代码库，而是从 Zuul 的代码库中获取。而 Zuul 则为测试任务合并了目标分支（master）的最新提交、本次变更的代码（变更 123）以及本变更所依赖的一系列变更，并通过 git + http 协议方式提供给 Jenkins 使用。测试任务不再需要关注代码准备过程，在后续章节讲解 pipeline 的功能时你将体会到 Zuul 这种服务所带来的优势。

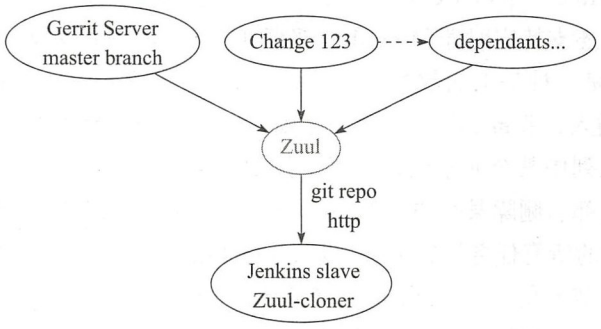


图 5-5　Zuul 为变更 123 提供代码准备服务

本节使用了较多的 Git、Gerrit 和 Jenkins 中定义的术语，详细内容请参考相关章节。

5.1.2　Zuul Server

Zuul Server 是 Zuul 的主要组件，有且仅有一个实例，不可扩展。只有当 Zuul Server 处

于运行状态时，Zuul 才可以进行各种操作。它从配置的远程系统（参见 5.3 节）的连接中接收事件，并把它们添加到 pipeline 列表中，调度执行队列中的事件并在事件执行后上报结果。Zuul Server 还提供了一个简单的 Web 客户端，可以浏览当前 pipeline 的信息。

图 5-5 中，除了 check、gate、post 这三个基本的 pipeline 外，还有 release（发布时执行）、periodic-weekly（每周执行）等自定义的 pipeline，在学习了后续的章节后，可以根据需要自行添加。

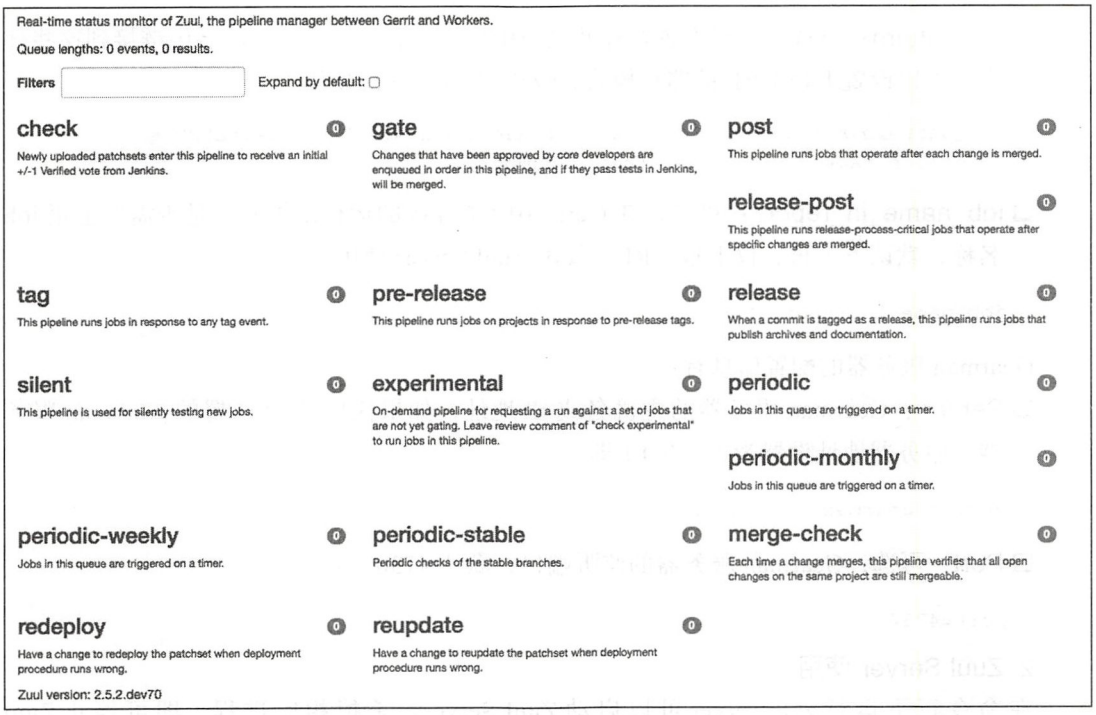


图 5-6　Zuul Web 客户端

1. Zuul Server 配置

Zuul Server 的配置在 zuul.conf 配置文件中，所有 Zuul 的组件都共享该配置文件。其中与 Zuul Server 相关的配置参数有：

- layout_config：layout.yaml 的配置文件的路径，必须存在，Zuul pipeline 在 layout_config 指定的文件中配置，仅由 Zuul Server 使用。

```
layout_config=/etc/zuul/layout.yaml
```

- log_config：可选，日志的配置文件，仅由 Zuul Server 使用。

```
log_config=/etc/zuul/logging.yaml
```

- pidftle：PID 文件，仅由 Zuul Server 使用。

```
pidfile=/var/run/zuul/zuul.pid
```

- state_dir：可选，Zuul 状态保存的目录，Zuul 所有的组件和命令都使用。

```
state_dir=/var/lib/zuul
```

- report_times：可选，在上报文本报告时，是否需要携带每个 job 已经运行的时间，仅由 Zuul Server 使用。

```
report_times=true
```

- url_pattern：可选，如果设置在外部系统中保存日志，并且在 Gerrit 中链接到这些日志，可以设置上报 URL 的路径模式，仅由 Zuul Server 使用。

```
url_pattern=http://logs.cibook.oz/{change.number}/{change.patchset}/
{pipeline.name}/{job.name}/{build.number}
```

- job_name_in_report：可选，在 Gerrit 中上报 job 的运行结果时，是否需要上报 job 名称，默认不上报，仅上报 URL，仅由 Zuul Server 使用。

```
job_name_in_report=true
```

Gearman 服务器的配置信息有：

- Server：Gearman 服务器的主机名或 IP 地址，如果使用 Zuul 内置的 Gearman 服务器，服务器地址设置为 127.0.0.1 即可。

```
server=gearman.cibook.oz
```

- Port：可选，Gearman 服务器的监听端口，默认 4730。

```
port=4730
```

2. Zuul Server 使用

在命令行下运行 zuul-server 可以启动 Zuul Server，杀掉相应进程，即可停止 Zuul Server 服务。zuul-server 命令行的基本使用方法如代码清单 5-1 所示。

代码清单 5-1　Zuul Server 命令行使用参考

```
usage: zuul-server [-h] [-c CONFIG] [-l LAYOUT] [-d] [-t] [--version]
Project gating system.
optional arguments:
  -h, --help show this help message and exit
  -c CONFIG  specify the config file
  -l LAYOUT  specify the layout file
  -d         do not  run as a daemon
  -t         validate layout file syntax
--version    show zuul version
```

可以通过 -d 参数让 Zuul 运行在非守护状态，这样可以快速检查配置文件中的问题。正常情况下，没有 -d 参数，Zuul 会以守护进程的方式运行。

非卖品，仅供非商业用途或交流学习使用

5.1.3　Zuul Merger

Zuul Merger 是一个独立组件，能通过 Gearman（参见 9.1 节）与 Zuul Server 通信。Zuul Merger 在其工作过程中进行一系列的 Git 操作，其目的是给每个 Zuul 准备进行的测试提供独立的代码合并服务，每一个将要测试的变更都单独地与目标分支的最新提交进行合并，以确保该变更是可合并的，同时保证，Zuul 所触发的测试能准确地表示合并变更后的结果。Zuul Merger 向测试工作节点（运行 Jenkins 任务的节点）提供 Git 代码库服务，而这些 Git 操作是 CPU 资源消耗密集型的任务，可以在不同的主机上运行多个 Zuul Merger 服务以进行负荷分担。

如果有多台 Zuul Merger 服务器，请确保 zuul_url 在每台运行 Zuul Merger 的主机上的设置是正确的，且各不相同。如图 5-7 所示，两个 Zuul Merger 对外的 Git 服务的地址分别是 http://a.cibook.oz/p 和 http://b.cibook.oz/p。

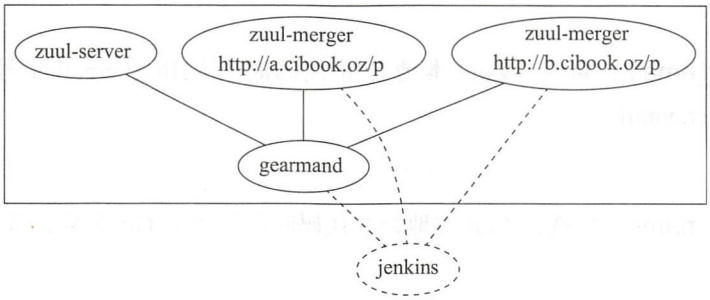


图 5-7　两个 Zuul Merger

在 Apache 服务器上进行如代码清单 5-2 所示的配置，Zuul Merger 就可以提供基于 http 的 Git 代码库服务。

代码清单 5-2　Apache 配置参考

```
SetEnv GIT_PROJECT_ROOT /var/lib/zuul/git
SetEnv GIT_HTTP_EXPORT_ALL
AliasMatch ^/p/(.*/objects/[0-9a-f]{2}/[0-9a-f]{38})$ /var/lib/zuul/git/$1
AliasMatch ^/p/(.*/objects/pack/pack-[0-9a-f]{40}.(pack|idx))$ /var/lib/zuul/git/$1
ScriptAlias /p/ /usr/lib/git-core/git-http-backend/
```

1. Zuul Merger 配置

Zuul Merger 也从 zuul.conf 文件中读取配置，需要对 gearman、gerrit、merger 和 zuul 等配置块进行配置。

（1）gerrit

Gerrit 服务器的配置信息。

❑ Server：Gerrit 服务器的域名或 IP 地址。

```
server=gerrit.oz
```

非卖品，仅供非商业用途或交流学习使用

- port：可选，Gerrit 服务器的 ssh 端口号，默认为 29418。

```
port=29418
```

- user：ssh 方式访问 Gerrit 服务器的用户名，添加 comments 和访问 stream-events。

```
user=cibook
```

- sshkey：ssh 方式访问 Gerrit 服务器所使用的密钥。

```
sshkey=/var/lib/zuul/ssh/id_rsa
```

（2）merger

Zuul Merger 运行相关的配置信息如下所示：

- git_dir：Zuul 克隆代码时保存在本地的路径。

```
git_dir=/var/lib/zuul/git
```

- git_user_email：可选，Zuul 本地合并代码时所使用的 email 账号，将传递给 git config user.email。

```
git_user_email=zuul@example.com
```

- git_user_name：可选，Zuul 本地合并代码时所使用的 Git 账号，将传递给 git config user.name。

```
git_user_name=cibook
```

- zuul_url：Zuul 对外提供的 Git repo 的地址，通常使用 https://zuul.cibook.oz/p 这种格式。

```
zuul_url=https://zuul.cibook.oz/p
```

（2）Zuul Merger 使用

在命令行下运行 zuul-merger 即可启动 Zuul Merger 服务。

Zuul 对于创建的 zuul ref 并不进行垃圾回收，如果需要清理这些引用，可以对每个 Git 库运行 tools/zuul-clear-refs.py 这个脚本，可以清理早于一定日期（默认为 360 天）的所有 Zuul 引用：

```
./tools/zuul-clear-refs.py /path/to/zuul/git/repo
```

5.1.4 Zuul Cloner

Zuul Cloner 是一个简单的命令行客户端，用于从 Zuul 引用所指定的项目库克隆（clone）文件。Zuul Cloner 优先使用 ZUUL_URL 环境变量指定的地址克隆代码，如果没有 ZUUL_URL 环境变量，就使用配置文件中的配置。

1. Zuul Cloner 配置

Zuul Cloner 重用了 Zuul 的参数，比如 ZUUL_BRANCH、ZUUL_REF 和 ZUUL_PROJECT。它将尝试从环境变量中获取这些变量，当然也可以通过命令行参数进行传递。命令行参数使

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

用 zuul 前缀，比如环境变量 ZUUL_REF 通过命令行参数 zuul-ref 来进行传递。

Clone map

默认情况下，Zuul Cloner 把项目克隆在 basepath 所指定的目录下，如果项目名称中包括字符 /，则说明项目名称使用了多级目录的层级形式，那么默认情况下 Zuul Cloner 会创建所有这些子目录。如果需要修改目标路径，可以通过 clonemap 来进行配置。配置文件使用 YAML 格式，通过 -m 参数指定。如下所示：

```
clonemap:
    - name: 'project'
      dest: '.'
```

如果需要去除一些子目录，可以通过正则表达式来修改。比如克隆项目 thirdparty/plugins/plugin1 时，希望去除 thirdparty 目录，直接克隆到目标目录 plugins/plugin1 下。如代码清单 5-3 所示，通过 \1 这个正则表达式的反向引用，工作目录就会只包括 ./plugins/plugin1 路径，而 project 项目则创建在当前目录下。

代码清单 5-3　修改克隆的目标目录

```
clonemap:
    - name: 'project'
      dest: '.'
    - name: 'thirdparty/(plugins/.*)'
      dest: '\1'
```

最终的映射关系如代码清单 5-4 所示：

代码清单 5-4　目标路径映射关系

```
project -> ./
thirdparty/plugins/plugin1 -> ./plugins/plugin1
```

2. Zuul Cloner 使用

通常，可以通过 zuul-cloner -h 来获得帮助，如代码清单 5-5 所示。

代码清单 5-5　zuul-cloner 命令行使用参考

```
# zuul-cloner -h
usage: zuul-cloner [-h] [-m CLONE_MAP_FILE] [--workspace WORKSPACE] [-v]
                   [--color] [--version] [--cache-dir CACHE_DIR]
                   [--branch BRANCH] [--project-branch PROJECT=BRANCH]
                   [--zuul-branch $ZUUL_BRANCH] [--zuul-ref $ZUUL_REF]
                   [--zuul-url $ZUUL_URL] [--zuul-project $ZUUL_PROJECT]
                   [--zuul-newrev $ZUUL_NEWREV]
                   git_base_url projects [projects ...]


Zuul Project Gating System Cloner.
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

位置相关参数：

- git_base_url：进行软件仓库克隆的基础地址。
- projects：需要克隆的项目列表。

可选参数：

- -m：指定 Clone Map 文件。
- --workspace：克隆的目标路径。
- --cache-dir：克隆的缓存路径。

项目相关参数：

- --branch：指定检出的分支，而不使用 Zuul 指定的分支，比如指定另一个分支用来测试客户端的兼容性。
- --project-branch：项目相关的分支，该参数比 --branch 具有更高的优先级。

zuul-cloner 以下面的次序尝试查找 Git 引用：

1）指定分支的 Zuul 引用

2）主分支的 Zuul 引用

3）指定分支的最新提交（tip）

4）主分支的最新提交

指定分支是下列分支之一：

1）项目指定的 branch（从 project_branches 参数中获得）

2）用户指定的 branch（从 branch 参数中获得）

3）ZUUL_BRANCH 指定的 branch（从 zuul_branch 参数中获得）

3. 克隆次序

当克隆项目库时，目标目录不可以存在，否则 Git 克隆会报错。由于 Zuul Cloner 支持一次克隆多个项目，如果在克隆父项目之前克隆子项目就会发生目录已存在的错误，比如：

```
zuul-cloner project/plugins/plugin1 project
```

这条命令在克隆 plugin1 项目时，创建了 project 目录，当克隆 project 项目时就会发生错误。调整次序，先克隆父项目再克隆子项目，就可以解决：

```
zuul-cloner project project/plugins/plugin1
```

4. 缓存项目库

--cache-dir 参数可以通过从本地代码库克隆，从而降低网络流量，但本地项目库可能不是最新的。如果 --cache-dir 参数存在，Zuul Cloner 将会先尝试从该目录克隆所有它需要处理的 Git 代码库，然后把克隆下来的代码库的 origin remote 的 URL 重置为 git_url_base，再进行上游项目库同步，这样本地代码库就会包含所有远端变更信息了。

--cache-dir 参数默认从环境变量 ZUUL_CACHE_DIR 中获取。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

5.1.5　Zuul 客户端

Zuul 有一个简单的命令行客户端，管理员可以用它来修改 Zuul 的运行行为。这个客户端程序需要可以访问 Gearman 服务器，比如：在 Zuul 服务器所在的主机就可以访问。

可以通过以下命令来获得更多的帮助：

```
zuul [enqueue|enqueue-ref|promote|show] --help
```

通过以下的命令，把 example_project 项目的 12345 变更的 1 号补丁放入 check pipeline 队列尾：

```
zuul enqueue --trigger gerrit --pipeline check --project example_project --change␣
→12345,1
```

通过以下的命令，把 12345 变更的 1 号补丁和 13336 变更的 3 号补丁调整到 check pipeline 队列的头部：

```
zuul promote --pipeline check --changes 12345,1 13336,3
```

下述命令可以显示当前正在进行的任务：

```
zuul show running-jobs
```

5.2　pipeline

Zuul 专注于解决如何合理安排变更的测试顺序，以便最大程度地并行测试这些变更。门控系统需要保证测试是基于变更合入到目标分支的最新提交后进行的。一个简单方法就是，一次只测试一个变更，如果通过测试就合入。这种方式看起来很好，但如果一个变更的测试时间非常长，开发人员就必须要等很长时间才能把变更合入到目标分支中。有些项目，测试任务需要花费好几个小时，而开发人员创建新变更和提交新补丁的速度远比测试和合入代码要快很多，这种简单方式就显得更加力不从心。

Zuul 支持两种 pipeline，分别是依赖型 pipeline 和独立型 pipeline。由于依赖型 pipeline 的原理比较复杂，而且是门控系统的最核心部分，下面将结合并行测试进行详细阐述。

5.2.1　并行测试

Zuul 依赖型的 pipeline 允许并行执行测试任务，同时确保变更得到正确的测试，对一个变更而言就像只有这个变更在测试。Zuul 假设所有的测试都能成功，这样所有的变更都可以被合并。如果有某个变更失败了，我们期望其他变更的测试任务能去除这个失败变更的代码，然后继续进行测试，就好像这个失败的变更根本没有被提交一样。理想情况下，期望尽可能多的变更被并行测试，尽可能多的变更得到合入；最差情况下，一次只能得到一个有效的测试结果（每个变更都测试失败，队列中第一个变更被认为测试失败，它之后的变更都需要重新测试，原理在后续部分详细描述）。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

举例来说，核心开发者短时间之内迅速批准了以下 5 个变更：

```
A, B, C, D, E
```

Zuul 会把这 5 个变更以它们被批准的次序添加到依赖队列中，并增加如下约束：

队列中每个后续变更都依赖它在队列中的前置变更。

这 5 个变更在队列中的依赖关系如图 5-8 所示，变更 A 位于队列首部，变更 E 位于队列尾部：

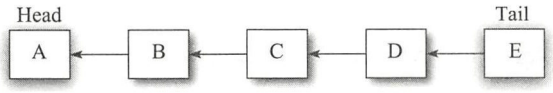


图 5-8　队列中 5 个变更的依赖关系

Zuul 随后并行启动这些变更的测试。需要强调的是：如果某个变更依赖于其他变更，Zuul Merger 会把它依赖的变更也包含在内，并假设它依赖的变更都能测试通过。本例中，B 的测试任务将会同时包含 B + A 变更的内容，这 5 个测试任务调度如下所示。

```
A 的测试任务：合入变更 A ，然后进行测试
B 的测试任务：合入变更 A 和 B ，然后进行测试
C 的测试任务：合入变更 A、B 和 C ，然后进行测试
D 的测试任务：合入变更 A、B、C 和 D ，然后进行测试
E 的测试任务：合入变更 A、B、C、D 和 E ，然后进行测试
```

因此，变更 A 测试只会包含 A，而忽略变更 B、C、D、E，如图 5-9 所示。

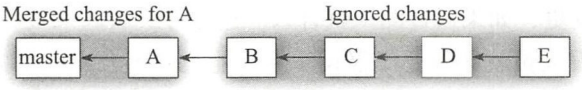


图 5-9　变更 A 的依赖关系

变更 E 的测试任务将包含整个依赖链上的变更，也就是包含了 A、B、C、D 和 E 所有这 5 个变更。变更 E 的测试任务假设变更 A、B、C 和 D 都能通过测试，如图 5-10 所示。

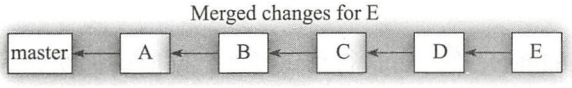


图 5-10　变更 E 的依赖关系

如果变更 A 和 B 都通过了测试，同时变更 C、D 和 E 都测试失败了，如图 5-11 所示。

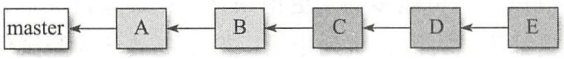


图 5-11　变更 A 和 B 测试通过

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

那么 Zuul 会依次合入变更 A 和 B，并从队列中移除，剩余的队列如图 5-12 所示。

图 5-12　去除变更 A 和 B 后的队列

由于变更 D 依赖于变更 C，无法确定变更 D 的测试任务失败是 D 本身有故障还是 C 导致。变更 E 的情况与 D 类似。但可以肯定的是，变更 C 测试失败且不依赖于队列中任何其他变更，因此 Zuul 会上报变更 C 测试失败，并把它从队列中删除，保留变更 D 和 E。

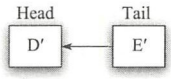


图 5-13　去除变更 C 后的队列

图 5-13 中这个队列就像两个新变更刚加入队列一样，Zuul 会重新启动针对变更 D 和 E 的测试。也就是说，变更 D 的测试任务只会包含变更 D 本身（如图 5-14 所示），而变更 E 的测试任务将会同时包含 D 和 E 的修改（如图 5-15 所示），但需要注意的是，此时新队列的 master 分支的最新提交已经发生变化，因为变更 A 和 B 已经合入到 master 分支。

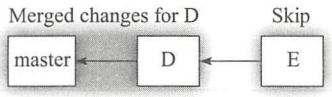


图 5-14　重启变更 D 依赖测试流程

在了解了 Zuul pipeline 的基本功能和原理后，不难看出 Zuul 的 pipeline 的调度管理和 Gerrit + Jenkins + 门控任务的普通门控系统有着较大的不同。

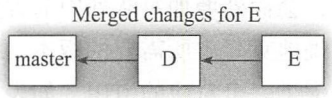


图 5-15　重启变 E 更依赖测试流程

- 普通门控系统没有队列管理功能，只能进行 master + 当前变更的并行测试。
- 普通门控系统无法保证在当前测试任务通过的情况下，合并代码后，目标分支的回归测试不受影响。
- 普通门控系统没有去除测试失败变更的自动回滚机制。

5.2.2　跨项目测试

当有多个项目互相之间紧耦合时，我们肯定希望进入队列的变更可以与当前队列中的其他关联项目的变更一起测试，因为关联项目的变更有可能引入了阻塞性功能，最终还会被合入主线分支。举例来说，服务器、客户端类型的项目，如果接口修改不能兼容，就演变为紧耦合的两个项目。当一个接口、功能变更同时涉及服务端和客户端的修改时，某一个变更测试通过不代表该功能与另一项目的最新版本也能测试通过。某些情况下，一个独立的变更甚至根本无法测试通过。

Zuul 不仅能解决同一个项目内的测试任务编排，也能用于跨项目的测试任务编排。这种跨项目，甚至多个项目的依赖关系，通过在同一个依赖型 pipeline 中注册相同的测试任务，就可以解决这种紧耦合的问题。

当一个变更进入队列时，Zuul 会同时创建该变更对其他项目的引用。举例来说，有一个名为 acme 的主项目和一个名为 plugin 的插件项目。定义一个测试任务 acme-tests，这两个项目的任何一个变更，都会触发这个测试任务运行，如代码清单 5-6 所示。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

代码清单 5-6 共享测试任务示例

```
pipelines:
    # 创建一个名为 gate 的依赖型 pipeline
    - name: gate
      # 依赖型 pipeline
      manager: DependentpipelineManager
projects:
    - name: acme
      gate:
            # acme 项目在 gate pipeline 中注册 acme-tests 测试任务
            - acme-tests
    - name: plugin
      gate:
          # plugin 项目在 gate pipeline 中注册同一个测试任务
          - acme-tests
```

当一个变更进入该 pipeline 队列时，Zuul 会为它创建一个引用，后续的变更进入队列时，会额外增加对队列中前置变更的引用。这样，依据依赖信息，队列中的每个变更都可以正确地克隆出所需要的版本。举例来说，3 个变更依次进入门控队列，Zuul 对于每个变更都分配了一个对应的引用，如表 5-1 所示。

表 5-1 Zuul 为变更序列创建引用

变更	项目	分支	Zuul 引用
Change 1	acme	master	master/Z1
Change 2	plugin	stable	stable/Z2
Change 3	plugin	master	master/Z3

由于变更进入的是依赖型的 pipeline，因此，与项目内的测试类似，Zuul 会为每个变更创建额外的引用，如表 5-2 所示。

表 5-2 Zuul 为变更序列创建额外引用

变更	项目	Zuul 引用	描述
1	acme	master/Z1	acme master + change 1
2	acme	master/Z2	acme master + change 1
	plugin	stable/Z2	plugin stable + change 2
3	acme	master/Z3	acme master + change 1
	plugin	stable/Z3	plugin stable + change 2
	plugin	master/Z3	plugin master + change 3

若要测试变更 3，需要克隆这两个代码库，同时把 Z3 引用应用于这两个项目库的每一个 project/branch 设定的匹配组合。在本例中，需要克隆：

❑ acme 项目：master 分支，应用 Z3 引用，包含了变更 1。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

❑ plugin 项目：master 分支，应用 Z3 引用，包含了变更 3 但不包含变更 2。

acme 项目的 master 分支和 plugin 项目的 stable 分支，如果不需要组合测试，则不需要克隆。

当测试任务需要克隆多个代码库，但队列中没有前置变更时，入队时就不会存在额外的引用变更，就只包含变更本身。

一个依赖型的 pipeline 可以根据需要有多个共享变更队列，相关联的项目共享一个变更队列，但不影响不相关的项目。一个独立型的 pipeline 不共享队列，但仍然可用于跨项目的交叉依赖测试。

依赖型 pipeline 的一个重要的特性是：它可以作为被不同项目触发的公共任务，它把这些项目看成是关联项目并共享同一个虚拟的变更队列。如果一个任务对两个项目进行集成，这两个项目自然就共享了一个虚拟的变更队列。如果一个第三方项目不使用这个任务，它就在这个虚拟队列的范围之外，变更也就会与这两个项目无关。

5.2.3　跨项目依赖

Zuul 允许用户通过在 git commit message 里通过特定的标注指定跨项目依赖。Zuul 的跨项目依赖（Cross-Repository Dependencies，CRD）的行为与 git 本身的做法是类似的，通过一个有向无环图（DAG）来实施。同一个库中一个变更单向依赖于另一个变更，而这种机制被 Zuul 借鉴用于不同项目库之间，准确地说，任何不同的 repo 之间的依赖都可以使用这种机制，而不限于是否是跨项目。如果变更 A 依赖 B，那么变更 B 就不可以依赖变更 A。

在 Git 的 commit message 中，包含关键字 Depends-On:<gerrit-change-id> 就可以向 Zuul 指示这种依赖关系。注意，需要使用完整的以字母 I 开头的 40 字节的变更号。通过 gerrit-change-id 可以从 Gerrit 系统中查询到具体的变更信息。具体看一个依赖的例子：

```
Depends-On: I835ddcb90c3b7b978eb80a0163d74faa254ab99a
```

1. 依赖型 pipeline

在处理 CRD 变更时，Zuul 会把这些具有依赖关系的变更放到同一个列队中进行序列化，也就是说，如果变更 A 依赖于变更 B，那么变更 A 和 B 会被加入到一个依赖型 pipeline 中，而且在队列中变更 B 在变更 A 的前面，如图 5-16 所示。

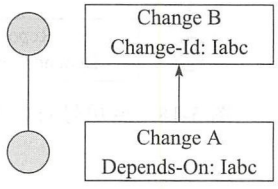


图 5-16　依赖型 pipeline 状态显示

如果变更 B 测试失败，那么变更 A 和变更 B 都会被从 pipeline 列队中清除。

2. 独立型 pipeline

相比于依赖型 pipeline，独立型 pipeline 要简单得多。所有变更相互之间都是独立的，整个 pipeline 没有依赖型 pipeline 采用的共享依赖图，然而对于每一个待测试的变更而言，Zuul 通过 Zuul 引用准备好它所有的依赖，然后进行测试。

当变更被添加到独立型 pipeline 队列时，变更本身的依赖（包括正常的来自于父提交的依赖和 CRD 变更）会像使用依赖型 pipeline 一样出现在本变更的依赖图中。也就是说，即使在独立型 pipeline 中，变更测试仍然是在合入它所有的依赖后进行的。通过这种机制，一些其他代码仓库中不能进行完整测试的变更在被依赖时被触发进行测试。

如果在 Web 客户端状态页面浏览这个依赖图，可以看到到依赖变更会被显示为灰色的圆圈，而实际被测试的变更显示为红色或绿色，取决于任务的测试结果，成功为绿色，失败为红色。

如图 5-17 所示，灰色（上）的圆圈说明变更仅用于其他变更的依赖。即使某个变更当前正在测试，但它作为依赖变更时，仍然显示为灰色；作为当前的测试变更时，它会显示为红色或绿色。

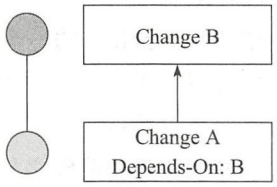


图 5-17　独立型 pipeline 状态显示

3. 多个变更依赖

一个 Gerrit 变更号可以指向多个变更（属于同一个项目的多个分支，甚至多个项目）。在这种情况下，Zuul 会把所有这些引用的变更都看作依赖。如果项目 A 的一个变更 A1 依赖于项目 B 的一个变更号 I123，但 I123 在项目 B 的两个分支都有变更，此时，测试 A1 这个变更时，项目 B 的两个变更都会被用于依赖。变更 A1 只有在项目 B 的这两个变更都已经合入的情况下才能合入，如图 5-18 所示。

当然，一个变更也可以依赖于多个 Gerrit 变更号。项目 A 可以同时依赖于项目 B 和项目 C 的变更号，而这只需要增加多行 Depends-On 即可。如图 5-19 所示，项目 A 的变更同时依赖于项目 B 变更号为 I123 的变更和项目 C 变更号为 Iabc 的变更。

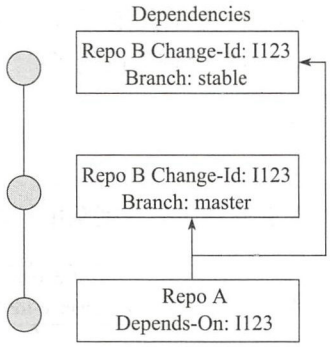


图 5-18　变更号对应多个变更分支

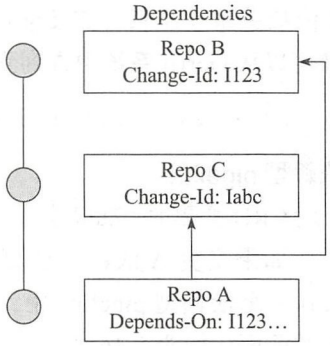


图 5-19　依赖多个变更号

5.3　连接器

Zuul 通过连接（connection）概念与多个不同系统进行通信。连接信息配置在 zuul.conf 文件中，在 layout.yml 文件引用这些配置。通过这种方式，Zuul 可以用一个连接从 Gerrit 服务器接收事件，却使用另一个连接上报结果，这两个连接使用不同的账号。

5.3.1　Gerrit

创建一个到 Gerrit 服务器的连接。

```
driver=gerrit
```

- server：Gerrit 服务器地址信息。

```
server=gerrit.cibook.oz
```

- port：可选，Gerrit 服务器端口。

```
port=29418
```

- baseurl：可选，Gerrit 服务器的 web 接口，默认设置为 https://<value of server>/。

```
baseurl=https://gerrit.cibook.oz/
```

- user：登录 Gerrit 服务器的 ssh 的用户名。

```
user=zuul
```

- sshkey：登录 Gerrit 服务器的 ssh 的密钥文件。

```
sshkey=/home/zuul/.ssh/id_rsa
```

- keepalive：可选，连接的保活时长，0 表示不保活。

```
keepalive=60
```

Zuul 通过 SSH 访问 Gerrit 服务器，因此需要给 Zuul 提供一个 Gerrit 服务器的 SSH 账号。如果这个账号没有配置密钥访问，需要为此用户在 Gerrit 服务器上添加公钥。比如，下面的命令在 gerrit.cibook.oz 服务器上，创建了一个 cibook 账号，并使用～ /id_rsa.pub 文件中的内容作为 SSH 的公钥。

```
cat ~/id_rsa.pub | ssh -p29418 gerrit.cibook.oz gerrit create-account --ssh-key \
    - --full-name CIBook cibook
```

并且需要在 Gerrit 服务器上为此用户进行合适的权限配置以满足进行门控系统的要求。举例来说，配置 Verified + 1/–1 的权限，用于指示该变更是否通过测试；配置 Submit 权限，以便于在 gate pipeline 通过时，把代码合入到主干分支。除此之外，还可以配置一些其他权限，这取决于需要 Zuul 完成什么任务，Zuul 的配置非常灵活，很容易进行适配。

注解　SSH 用户必须具有 stream-events 的权限，否则 Zuul 无法收到 Gerrit 触发事件。

5.3.2　SMTP

创建一个到 SMTP（Simple Mail Transfer Protocol）服务器的连接。

```
driver=smtp
```

- server：SMTP 服务器的主机名或 IP 地址。

```
server=localhost
```

- port：可选，SMTP 服务器端口号。

```
port=25
```

- default_from：邮件报告的默认发件人的邮件地址，该参数可以在 pipeline 中重定义。

```
default_from=zuul@cibook.oz
```

- default_to：邮件报告的默认收件人的邮件地址，该参数可以在 pipeline 中重定义。

```
default_to=you@cibook.oz
```

5.4 触发器

Zuul 主要支持 Gerrit 作为触发器。由于 Zuul 采用模块化设计，易于进行扩展，因此很容易支持其他触发和上报系统。

5.4.1 Gerrit

Zuul 可以和标准的 Gerrit 系统对接，通过在 SSH 连接上执行 gerrit stream-events 标准命令来获得事件通知，上报也在该接口上进行。

如果使用 Gerrit 2.7 或者更新的版本，需要确保配置给 Zuul 的用户所在组带有 Stream Events 权限，否则 Zuul 将无法执行 gerrit stream-events 命令。

参见图 5-20，给 Zuul 配置的访问 Gerrit 的账号所在的组需要在 Projects → Access → Global Capabilities → Stream Events 标签下进行配置。

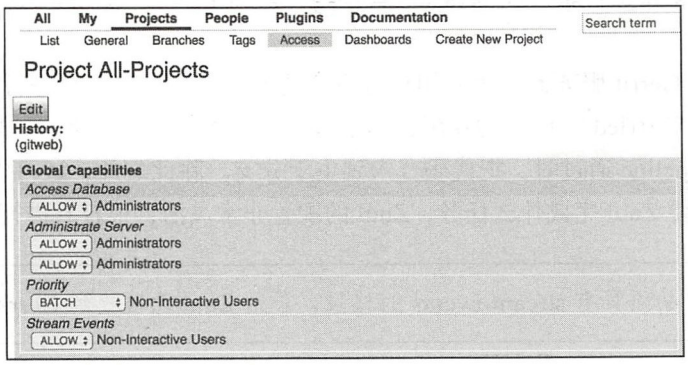


图 5-20 Gerrit Stream Events 权限配置

Gerrit 驱动的连接名称可以支持多种事件和以下的参数：

- event：Gerrit 系统的事件名称，比如 patchset-created、comment-added、ref-updated。该字段是正则表达式。
- branch：事件对应的分支名称，比如 master 对应 master 分支。该字段是正则表达式，且可以有多个列表。
- ref：对于 ref-updated 事件，branch 参数无效，但提供 ref 参数。当前，Gerrit 在某种程度上来说对于 ref 的处理有些特殊，对路径指定裸引用（在 ref 中不包含前缀，如 master），对其他使用全路径引用（如 refs/tags/foo）。Zuul 使用 Gerrit 提供的参数进行精确匹配。该字段是正则表达式，且可以有多个列表。
- approval：仅用于 comment-added 事件中，用于匹配该变更的评审通过信息，比如 code-review:2 用于匹配代码评审的记录中是不是含有 + 2。该参数可以包含多个变更通过的信息列表。
- email：该字段适用于任何事件。该字段使用正则表达式对操作者的 Gerrit 账号使用 email 进行匹配。如果希望对多个 email 进行匹配，必须使用 YAML 的列表形式。请确保使用非贪婪的匹配规则，并且对于分隔域名的 . 号需要使用转义符 \。比如：email: ^.*?@example\.org$，域名 exapmle.org 中的 . 中需要使用转义符。
- username：该字段适用于任何事件。该字段使用正则表达式对操作者的 Gerrit 账号使用 username 进行匹配。如果希望对多个 username 进行匹配，必须使用 YAML 的列表形式。请确保使用非贪婪的匹配规则，并且对于 . 号需要使用转义符 \。比如：username:^jenkins$。
- comment：仅用于 comment-added 事件。它使用一个正则表达式列表在 comment 字符串中搜索。如果任一正则表达式匹配了 comment 字符串，该触发器就匹配了。比如：comment:retrigger 仅当变更的 comment 文本中包含有 retrigger 字符串时匹配。
- require-approval：该字段适用于所有事件。它需要变更的当前补丁满足一定的预定义条件（可以通过事件增加）。它与 5.6.1 节 pipeline 的 require 使用相同的语法描述。
- reject-approval：和 require-approval 使用相同的格式，只是当条目满足时，拒绝评审通过。

5.4.2　Timer

Zuul 也支持简单时间触发，它支持基于 cron 定时系统服务的指令格式触发 pipeline 中的任务。时间触发不需要任何的连接或驱动。同时，时间触发器也可以使用一个 timer 列表作为触发器。时间触发会把在配置文件中定义的时间信息作为一个事件放到每个项目的 pipeline 中。任何与该 pipeline 相关的任务，当事件发生时就会被触发执行。

timer 使用 5 个参数，与 cron 系统服务是类似的：比如，0 0 * * * 表示在每天凌晨 00:00 分运行该任务；10 0 * * 1 表示在每周二上午 10:00 执行该任务，具体可以参考[⊖]。需要注意的

⊖ https://docs.acquia.com/article/cron-time-string-format。

是，第 5 个参数取值范围是 0-6，1 在 crontab 表示周一，但在 zuul 中表示周二，依此类推。

5.4.3 Zuul 内部事件

Zuul 基于内部的动作会产生一些事件，也支持多个事件列表。Zuul 的事件不需要专门的连接或驱动，它们以列表的方式使用。

- event：事件名称，当前支持如下事件：
  - project-change-merged：当 Zuul 合并一个变更到目标分支时，它给项目每一个未完成的变更产生一个该事件。
  - parent-change-enqueued：当 Zuul 把一个变更加入到 pipeline 队列时，它给该变更的每个子变更产生一个该事件。
- pipeline：只对 parent-change-enqueued 事件有效，是父变更被放到队列时的 pipeline 名称。
- require-approval：适用于任何事件，需要满足的通过条件，同第 5.6.1 节 pipeline 的 require 使用相同的语法描述。
- reject-approval：同 requre-approval，二者只是作用相反，满足条件的情况下拒绝加入队列。

5.5 报告器

Zuul 以配置的方式上报测试结果，比如 build pipeline 成功运行之后，在 Gerrit 上返回一个成功（+ 1/+ 2）的评审结果。

Zuul 报告器一般有三种处理阶段：开始、成功或失败。每个阶段都可以有多个上报，比如在测试开始时重设 Gerrit 的 Verified标签，并且发送一封邮件通知。

5.5.1 Gerrit

与从 Gerrit 服务器获取通知事件一样，Zuul 也使用 ssh 协议向 Gerrit 服务上报结果。

传递给 Zuul 的参数字典被用于向 Gerrit 服务器上报结果的 gerrit review 命令。字典参数的值如果是布尔值 true，表示该参数必须存在，但不需要一个数值。比如，verified:1 表示 gerrit review--verified 1，submit:true 则表示 gerrit review--submit。

5.5.2 SMTP

SMTP 提供简单的 email 上报功能。如果需要此功能，需要配置 SMTP 驱动，具体参见 5.3 节。

使用该报告器，需要在 zuul.conf 中配置 SMTP 服务器及默认的 to/from信息。每个 pipeline 都可以通过参数来重新设置主题（subject）、收件人（to）和发件人（from）参数，如代码清单 5-7 所示。

代码清单 5-7　pipeline 定义示例

```
pipelines:
    - name: post-merge
        manager: IndependentpipelineManager
        source: my_gerrit
        trigger:
            my_gerrit:
                - event: change-merged
        # 在 pipeline 成功时发邮件给 you@cibook.oz
        success:
            outgoing_smtp:
                to: you@cibook.oz
        # 在 pipeline 失败时发送主题为变更 {change} 失败的邮件给 you@cibook.oz
        # 且发件人为 alternative@cibook.oz
        failure:
            internal_smtp:
                to: you@cibook.oz
                from: alternative@cibook.oz
                subject: Change {change} failed
```

本代码清单 5-7 中，在 post-merge 这个后处理 pipeline 中，执行成功后使用 outgoing_smtp 连接器向 you@cibook.oz 发送成功邮件；执行失败后使用 internal_smtp 连接器，向 you@cibook.oz 发送主题为 Change{ 变更号 }failed 的邮件，且发件人是 alternative@cibook.oz。

5.6　配置指导

OpenStack 社区的开发全部是基于 Gerrit 代码评审系统进行的，Zuul 与 Gerrit 关系密切，Zuul 的不少流程是为 Gerrit 定制开发的，而 Gerrit 为适应 Zuul 和社区流程的需要，也需要做一些定制配置（无须修改版本），比如增加 workflow 标签。到目前为止，Zuul（V2）只支持 Gerrit 这一种代码管理工具。在浏览具体的参数之前，先看一下 Gerrit 系统上可以提供的信息，这些信息在后面的参数说明中将要使用到。

Gerrit 的详细说明请参见本书 3.2 节，这里重点描述与 Zuul 相关的内容，如图 5-21 所示。

对于图 5-21 中一些重要内容进行如下说明：

1）Gerrit 系统中提供了 3 个标签（Label）属性：Code-Review、Verified 和 Workflow，其中前两个是 Gerrit 系统内置的 Label，第三个 Workflow 是 Openstack 社区主要使用的一个 Label，主要目的是为了完成代码提交的自动化，减少人工操作。当有两个 +2（Clark Boylan 和 Jens 位 Harbott）的 Code-Review 的情况下，核心贡献者（Jens Harbott）把 Workflow 设置为 +1，这就触发了进入 gate pipeline 流程。

2）547041 为变更号，引用该变更的 URL 是 https://review.openstack.org/#/c/547041。

3）项目名为 openstack-infra/project-config，该名称也是 Gerrit 中的 repo 名称。

4）该项目使用了 check 和 gate 两个 pipeline，每个 pipeline 都只有一个任务，并且这两

个 pipeline 共享了同一个测试任务 openstack-zuul-jobs-linters。

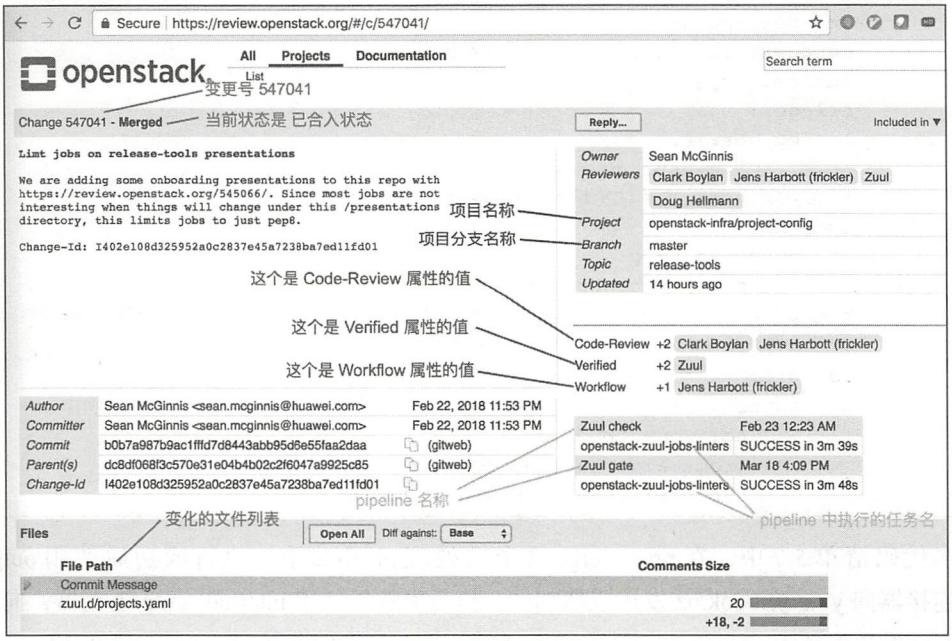


图 5-21 Gerrit 评审单状态

5）本次变更只修改了一个文件，即 zuul.d/projects.yaml。

图 5-22 是 Gerrit 系统中的 Comments 相关的内容。

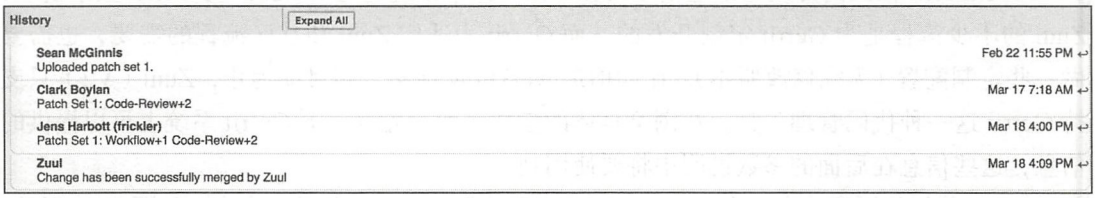


图 5-22 Gerrit 评审意见

从图 5-22 中可以看到，在 Jens Harbott 同时发表了 Workflow + 1 和 Code-Review + 2 的 Comments，大约 9 分钟后，Zuul 完成了代码提交，核心贡献者没有参与 Gerrit 的 Submit 过程。

Zuul 有三个配置文件，其中前两个是必需的：

- zuul.conf：使用 INI 文件格式，提供了 Zuul 运行所需要的绝大部分配置信息。
- layout.yml：pipeline 运行所需信息。
- logging.conf：可选的 python 日志配置文件。

所有的 Zuul 组件都可从 /etc/zuul/zuul.conf 文件中读取配置，当然也可以从命令行指定其他文件。同时需要为 Zuul 准备一个 Gerrit 库的账号用于 Zuul 访问 Gerrit 代码库。详细配

置参见 5.3 节。

zuul.conf 文件示例参见代码清单 5-8。

代码清单 5-8　zuul.conf 配置文件

```
[zuul]
layout_config=/etc/zuul/layout.yaml
[merger]
git_dir=/git
zuul_url=http://zuul.cibook.oz/p
[gearman_server]
start=true
[gearman]
server=127.0.0.1
[connection gerrit]
driver=gerrit
server=gerrit.cibook.oz
port=29418
baseurl=http://gerrit.cibook.oz
user=cibook
sshkey=/home/zuul/.ssh/id_rsa
```

本章前面的章节已经详细描述过 zuul.conf 中各参数的含义，此处不再详述，后文重点讲解 layout.yaml 相关的关键参数。

5.6.1　pipeline

Zuul 有许多 pipeline 的参数，当某个 pipeline 与 Gerrit 的某个事件匹配时，这个事件就会被添加到 pipeline 的队列，与这个 pipeline 相关的 job 就会被调度执行。当该事件触发 pipeline 的所有 job 都执行完成时，Zuul 会向 Gerrit 报告测试执行结果。本章描述 pipeline 的基本原理时，也说明过 Zuul 不存在预定义的 pipeline，因此可以根据需要在 layout.yml 中定义任意的 pipeline，同时 pipeline 的数目也没有限制。Zuul pipeline 这种灵活的机制可以适应任何实际工作流的需要。

- name：pipeline 的名称，后续在定义 job 时需要引用，这个与 jjb 的宏定义的使用是类似的。
- source：给 Zuul 提供触发事件、变更控制的系统，Zuul 通过驱动与这些系统进行连接，可以给 Zuul 添加不同系统的驱动，可扩展性很强，不过当前只支持 Gerrit 驱动，该参数是必选参数。
- success-message：可选，当所有具有投票（voting）权限的 job 执行成功之后，Zuul 会向 Gerrit 上报成功结果，默认是 Build Successful。

```
success-message=Build Successful.
```

- failure-message：可选，当任何一个具有投票权限的 job 执行失败之后，Zuul 上报执行失败的结果，默认值是 Build failed.。

```
failure-message=Build failed.
```

- merge-failure-message：可选，当一个变更的当前状态无法合并（Merge）到目标分支时（说明存在变更冲突），Zuul 会向 Gerrit 上报合并失败的消息。

```
merge-failure-message=Merge failed.
```

- footer-message：对测试结果提供额外信息，比如，对 CI 系统提供额外的调试和联系方式的信息还是比较有用的。
- manager：两种类型的 pipeline 的管理器，IndependentpipelineManager 和 DependentpipelineManager 分别对应本章前面讲解的独立型 pipeline 和依赖型 pipeline。
- trigger：一个 pipeline 至少需要提供一个触发器信息，这些触发器不是排它的，即一个 pipeline 可以使用多个触发器，一个触发器也可以在多个 pipeline 中使用，在这些 pipeline 中的行为也是独立、互不影响的。

  每个在 source 中定义的驱动，都提供了不同触发器选项，详细信息参见 5.4 节。
- require：可选，进入 pipeline 需要满足的预定义条件，而不管该条件是如何进入，无论是通过触发器触发还是自动依赖引入，本参数定义的条件必须满足，否则事件不能进入 pipeline 队列。

  该参数定义的子参数较多，见下面详细说明：
  - approval：对于变更要求批准的信息满足一定的条件，它本身也有多个子参数，所有这些子参数都是可选的，且可以组合使用，但如果使用，则需要同时都满足所有出现的子参数。

    username：批准变更的用户名，使用正则表达式表示。

    email：批准变更的邮箱，使用正则表达式表示。

    older-than：变更批准需要经历的时间，通常支持 w（weeks）、d（days）、h（hours）、m（minutes）、s（seconds）等后辍，比如 48h 表示 48 小时，2d 表示 2 天。

    newer-than：和 older-than 使用相同的格式，要求批准必须在多长的时间之内，这两个参数一般不使用。

    除上述参数外，所有其他参数都被解释为 review 相关的参数。比如 verified:1 要求在 Gerrit 系统中 Verified 列存在 + 1 这个数据，取值可以是一个单一值，也可以是一个列表，比如 verified:[1，2] 要求或者 + 1 或者 + 2 的投票存在。
  - open：表示进入队列的变更状态是未合入状态（open）还是合入状态（closed），布尔值。
  - current-patchset：表示进入队列的变更是否需要当前最新的补丁，布尔值。在 Gerrit 系统上，可以浏览一个变更的所有补丁，甚至可以在 Gerrit 上触发基于某个历史变更的测试，比如使用 recheck 来触发。当 current-patchet 设置为 True 时，Zuul 忽略除最新补丁之外的其他触发请求；当 current-patchet 设置为 False 时，变更的任何一个历史补丁都可以触发测试，在不回退版本的情况下，就可以验证某

个历史的补丁的效果，这个特性在某些情况下非常有用。

- status：表示触发器上报事件的当前状态的字符串，比如 NEW or MERGED。

- reject：用于阻止变更进入 pipeline 队列，与 require 作用正好相反，使用与 require 类似的条件。
- approval：和 require 参数的 approval 一样，使用多个子参数。举个例子，如果存在 –1 或 –2 的投票结果，则拒绝入队：

```
reject:
approval:
- code-review: [-1, -2]
```

- dequeue-on-new-patchset：通常，一个当前正在 review 的变更提交了一个新补丁，如果这个变更已经在 pipeline 之中，那么会从 pipeline 中去除该变更的旧条目，并取消所有与该旧条目相关的、包括依赖该变更的测试任务。当然，可以通过设置参数来修改这个默认行为，该参数为布尔值。
- ignore-dependencies：Zuul 会尝试把当前变更的所有依赖变更放到 pipeline 中该变更的前面，这样所有这些依赖和该变更可以被一起测试。该参数是布尔型参数，默认取值是 false，该参数只对独立型 pipeline 有效，把该参数设置为 true，可以完全取消依赖。
- success：可选，描述所有测试任务完成之后，Zuul 如何向 Gerrit 上报结果。
  - 如果参数不存在，所有任务运行成功之后，Zuul 什么也不做，甚至不向 Gerrit 上报任何信息。
  - 如果参数存在，将会使用一个上报插件列表来上报测试结果，每个条目代表了一个插件连接名称，取决于为此连接提供的驱动。

  参见 5.5 节的详细描述。
- failure：使用与 success 一样的语法和参数，只是说明在测试失败时，Zuul 如何处理。
- merge-failure：使用与 success 一样的语法和参数，只是说明在合并变更失败时，Zuul 如何处理。如果该参数不存在，failure 上报器会用于通知不成功的合并操作消息。
- start：使用与 success 一样的语法和参数，只是说明合并在被添加到 pipeline 队列时，Zuul 如何处理。比如：可以重置 Gerrit 系统中 Verified 条目的值。
- disabled：使用和 success 一样的语法和参数，只是说明变更的目标 pipeline 被禁用时，Zuul 如何处理，参见 disable-after-consecutive-failures。
- disable-after-consecutive-failures：可选，如果设置了该参数，当太多的变更接连失败，pipeline 会进入禁用状态（disabled）。当连续的失败次数超过该值时，Zuul 除给 disabled 上报器上报外，会停止给任何其他上报器上报，无论是 success、failure 还是 merge-failure。当 pipeline 处于禁用状态时，start 上报器也不会上报。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- precedence：可选，指示对于不同的 pipeline 的任务如何按优先级进行调度。每个 pipeline 都有一个优先级，具有更高优先级的 pipeline 中的任务比较低优先级的任务得到更早的执行。该字段的取值范围是 high、normal 或 low。

```
precedence=normal
```

- window：可选，仅用于依赖型 pipeline。Zuul 可以像 TCP 一样进行流控。pipeline 队列会有一个窗口值，只有窗口可用时（即大于 0），变更对应的任务才会被执行，窗口大小的初始值被设置为该配置值，配置值是一个正整数，配置为 0 会禁用流量控制。

```
window=20
```

- window-floor：可选，仅用于依赖型 pipeline。用于指示 window 的最小值，正整数。

```
window-floor=3
```

下面分别了解一下在 OpenStack 中三个最常用的 pipeline：check、gate 和 post。先了解一下 check pipeline 的实际例子，pipeline 根据需要使用相关的参数。

代码清单 5-9　Check pipeline 示例

```
    pipelines:
        # 定义了 check pipeline
        - name: check
          description: Newly uploaded patchsets enter this pipeline to receive an
initial +/,!-1
                       Verified vote from Jenkins.
          # 所有带有 voting 权限的任务成功执行后，向 Gerrit 增加的 comment 消息
          success-message: Build succeeded (check pipeline).
          # 任一有 voting 权限的任务执行失败后，向 Gerrit 增加的 comment 消息
          failure-message: Build failed (check pipeline). For information on how to,
!proceed, see
                           http://docs.openstack.org/infra/manual/developers.html#
automated-
    ,!testing
        # 独立型 pipleline 管理器
        manager: IndependentpipelineManager
        # 使用 Gerrit 作为变更的数据源
        source: gerrit
        # normal 等级的 pipeline
        precedence: normal
        # 进入该 pipeline 的条件，要求处于 open 状态且是当前补丁
        require:
            open: True
            current-patchset: True
        # Gerrit 类型的触发器，与 source 保持一致
        trigger:
            gerrit:
                # 有新的补丁（patchset）被创建时
                - event: patchset-created
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
            # 或者补丁被恢复时
            - event: change-restored
            # 或者有新的评论，且包含 recheck 时，comment 中使用了正则表达式表示方式
            - event: comment-added
              comment: (?i)^(Patch Set [0-9]+:)?( [\w\\+-]*)*(\n\n)?\s*recheck
            # 或者有新的评论，且来自于用户名含有 jenkins 的 Verified 值为 -1 或 -2 且 workflow 被置为 1
            - event: comment-added
              require-approval:
                    - verified: [-1, -2]
                      username: jenkins
              approval:
                    - workflow: 1
    # 执行成功后，在 Gerrit 设置 Verified 的值为 1
    success:
        gerrit:
            verified: 1
    # 执行失败后，在 Gerrit 设置 Verified 的值为 -1
    failure:
        gerrit:
            verified: -1
```

对代码清单 5-9 中 check pipeline 的一些重要信息说明如下：

1）使用独立型（IndependentpipelineManager）pipeline，优先级为 normal。

2）进入队列时，变更需要处于 Open 状态，且只有该变更的最新补丁才可以触发测试。

3）有 4 种 Gerrit 事件可以触发该 pipeline，包括新的补丁、变更被恢复或者有新的评审意见。其中 recheck 这个评论方式非常有用，通过正则表达式定义了一个包含 recheck 的字符串（这个字符串格式需要参考 Gerrit 系统 comments 的消息定义）。如果依赖项目或基础设施出现问题，在这些问题被解决后，变更本身并不需要修改，但需要有能力重新触发测试，这个示例就是为了解决该问题。在 Gerrit 界面的 Reply 消息框中输入 recheck 字符串即可触发任务重新执行。当然，这个正则表达式可以定义为任何其他方便使用的格式，只是 recheck 比较通用。

4）如果执行成功，在 Gerrit 上设置 Verified 值为 1；如果执行失败，在 Gerrit 设置 Verified 的值为 -1。

下面来看一下 gate pipeline 的实际例子，与 check pipeline 相同的内容在本例中将略过。

代码清单 5-10　Gate pipeline 示例

```
- name: gate
    ...
    # 依赖型 pipeline 管理器
    manager: DependentpipelineManager
    # 使用 Gerrit 作为变更的数据源
    source: gerrit
    # high 等级的 pipeline
    precedence: high
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
        # 进入该 pipeline 的条件，要求：
        # 处于 open 状态且是当前补丁，
        # 并且来自于用户名包含 cibook 或者 openzeroci 的 +1/2 的 Verified 结果，
        # 同时 workflow 是 1
        require:
            open: True
            current-patchset: True
            approval:
                - verified: [1, 2]
                    username: (cibook|openzeroci)
                - workflow: 1
        # Gerrit 类型的触发器，与 source 保持一致
        trigger:
            gerrit:
                # 有新 workflow 为 1 的评论
                - event: comment-added
                    approval:
                        - workflow: 1
                # 或者有新的 Verified 的值为 1，且用户名包含 cibook 或 openzeroci
                - event: comment-added
                    approval:
                        - verified: 1
                    username: (cibook|openzeroci)
        # 任务开始化，把 Gerrit 上的 Verified 的值设置为 0
        start:
            gerrit:
                verified: 0
        # 所有带有 voting 权限的任务执行成功后，把 Gerrit 上的 Verified 的值设置为 2，并且把变更
提交到
        # 目标分支
        success:
            gerrit:
                verified: 2
                submit: true
        # 任一带有 voting 权限的任务执行失败后，把 Gerrit 上的 Verified 的值设置为 -2
        failure:
            gerrit:
                verified: -2
        # 最小窗口大小
        window-floor: 20
        window-increase-factor: 2
```

对代码清单 5-10 中 gate pipeline 的一些重要信息说明如下：

1）使用依赖型 pipeline，优先级为 high。

2）进入 pipeline 的条件除了 check pipeline 使用以外，还增加了如下约束：Verified 标签存在来自于用户名包含 cibook 或 openzeroci 的 + 1 或 + 2 的验证结果，同时 Workflow 标签值是 1；结合 Gerrit 系统中每个标签的权限都是受控的，这个依赖组合相对来说比较复杂也比较严格，这也正是 Zuul 严格把关的体现。一般而言，只有 CI 账号返回的 Verified 的结

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

果才是可信的、有效的，也可以有效地防止误操作。

3）与 check pipeline 不同的是，gate pipeline 一般都是只由 comment 事件触发。比如：workflow 标签被设置为 1，或者包含用户名为 cibook 或 openzeroci 的用户把 Verified 标签设置为 1。

4）注意，在 pipeline 开始时，Verified 标签会被 pipeline 重置为 0。

5）如果测试任务执行成功，那么 pipeline 会把代码提交合并到主分支。下面来看一下 post pipeline 的实际例子，如代码清单 5-11 所示。

代码清单 5-11　Post pipeline 示例

```
- name: post
    # 独立型的  pipeline 管理器
    manager: IndependentpipelineManager
    source: gerrit
    # 低  (low) 优 先 级
    precedence: low
    # Gerrit 类型的触发器，与  source 保持一致
    trigger:
    .gerrit:
        # ref-updated 消息，且  ref 是包含  ^refs/heads/.*$ 正则表达式
        - event: ref-updated
          ref: ^refs/heads/.*$
```

从代码清单 5-11 中可以看到，post pipeline 监听 ref-updated 消息，只关心 Git 引用是 ref/head/ 相关的变化，亦即代码合入到目标分支的消息。这可以用于文档发布，以及触发后续的自动化安装、测试和部署等流程。

5.6.2　Jobs

job 是 layout.yml 配置文件中的可选部分，用于设置任务的属性，这些属性与 project 小节没有相关性。举例来说，一个任务运行失败后，希望返回自定义消息，可以在本节进行配置；或者当变更的内容满足一定的条件时才触发任务的执行。否则，如果没有配置，Zuul 可以从 project 部分加载任务相关的信息。

- name：Jenkins job 的名称，该名称参见 4.1 节详细的定义。该字段使用正则匹配，所有匹配的任务都会应用该规则，必须存在。本节除 name 外的所有其他字段都是可选字段。
- queue-name：Zuul 会把多个使用相同 Jenkins 任务的项目合并，让他们共享依赖型 pipeline 管理器。给这些队列进行命名，在上报队列统计数据时将会非常有用。当然，如果不提供名字，Zuul 会自动给每个队列进行命名，但这个名字将会非常长，而且会根据队列中的项目变化而动态修改。如果给任务指定了一个队列名称，Zuul 会将该名称用于共享队列，而不是自动产生一个新的。如果有多个任务需要使用同一个共享队列，但队列的名称不同，这肯定是错误的配置。
- failure-message：同 pipeline 中的配置。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- success-message：同 pipeline 中的配置。
- failure-pattern：如果任务执行失败，上报给 Gerrit 系统中的 URL 的格式。如果不指定，默认是构建 URL 或者是在 zuul.conf 中定义的 url_pattern。
- success-pattern：同 failure-pattern。
- hold-following-changes：用于指示依赖型 pipeline 队列中的变更是否需要等待本任务成功执行之后再执行其他任务的布尔值。如果先运行一个时间非常短的任务，但该任务可以更早预测一个需要长时间运行任务的失败，这毫无疑问能减少 Zuul 发起的、最终又会被取消的任务数目。在这种场景下，少量的短任务能更有效地利用测试资源。另一方面，如果把这种机制用在一个需要长时间运行的任务上，这将使得 Zuul 的主要特性，并行测试依赖变更的目标落空，该特性需要谨慎使用。
- mutex：从名称上可以看出该字段的含义，与程序语言的互斥量是相同的意思。这是任务可以观察的一个字符串的名字，同一时间，只能有一个使用该名字的任务进入队列。注意，该参数作用于所有的 pipeline。
- branch：该任务在什么分支上才会执行，正则表达式参数。
- files：只有当变更涉及列表中至少一个文件的变化时，包括增加、删除和修改，才会执行该任务。正则表达式参数，支持列表。
- skip-if：在所有可选参数都满足的情况下，该任务就不会被执行。如果有多个候选集，任何一个候选集满足，该任务就不会执行。在一个项目有多个子目录、多个测试任务，并且当测试任务只与部分子目录下的文件有关，与其他子目录下的文件没有关系，该参数就非常有用，可以有效地减少并发测试任务的数目。
  - all-files-match-any：一个正则表达式列表，用于与变更相关联的文件名进行比较。如果一个变更中所有的文件名都至少满足列表中一个表达式时，则认为该条件匹配。其中 /COMMIT_MSG 会一直被使用，因此不需要包含。但合并的提交（merge commits）是个例外，即使 /COMMIT_MSG 不匹配，任务也不会被跳过。如果是合并提交，假设涉及修改的文件是不可预测的，CI 的任务必须被执行。

  如代码清单 5-12 所示的例子，在以下两种情况下，测试任务被忽略：
  - openstack/neutron 项目的 stable/juno 分支并且所有修改的文件都在 neutron/tests 或 tools 子目录下。
  - 所有修改的文件都在 doc 子目录下或者都是以 rst 结尾的文件。

代码清单 5-12　Job skip-if 配置示例

```
jobs:
    - name: check-tempest-dsvm-neutron
        skip-if:
            # 如果是 openstack/neutron 项目，且是 stable/juno 分支并且
            # 所有修改的文件都在 neutron/tests 子目录或 tools 子目录下
            # 那么本任务就不会被执行
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
-     project: ^openstack/neutron$
      branch: ^stable/juno$
      all-files-match-any:
          - ^neutron/tests/.*$
          - ^tools/.*$
      # 如果所有修改的文件都在  doc 子目录下或者都是以  rst 结尾的文件
      # 那么本任务就不会被执行
      - all-files-match-any:
          - ^doc/.*$
          - ^.*\.rst$
```

- voting：表示该任务是否需要投票的布尔值，用于在 Gerrit 系统上设置 Verified 标签的值。

```
voting=true
```

- attempts：Zuul 尝试启动一个任务的次数，如果该次数达到，Zuul 会上报 RETRY_LIMIT。

```
attempts=3
```

5.6.3　Projects

本小节的参数说明每个项目（对应 3.2 中的 repo 库）在每个 pipeline 中运行哪些任务。它包含一个列表，如代码清单 5-13 所示。

代码清单 5-13　Project 配置示例

```
- name: example/project
    check:
        # 当  project-merge 执行成功后，才会执行其他 3 个任务
            - project-merge:
            - project-unittest
            - project-pep8
            - project-pyflakes
    gate:
        - project-merge:
            - project-unittest
            - project-pep8
            - project-pyflakes
    post:
        - project-publish
```

- name：Gerrit 中 repo 库的名称。
- merge-mode：可选，当把变更合并到目标分支时采用的策略，支持以下的取值：
  - merge-resolve：等价于“git merge -s resolve”。这相当于当 Gerrit 系统的合并模式配置为尽可能合并并且自动解决冲突时，Gerrit 系统对一个项目所进行的合并操作，这也是默认值。
  - merge：等价于“git merge”。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- cherry-pick：等价于“git cherry-pick”。

这些参数后面就是前面定义的 pipeline 的引用，如果该 pipeline 下没有任务，那该 pipeline 不需要被引用或出现。在每个 pipeline 的配置小节中，列出每个需要执行的任务列表。如果某个测试任务名称被用于哈希字典的 key（YML 语法），那么只有当这个名为 key 的任务执行成功后，其他任务才执行。在上面的例子中，只有当 project-merge 执行成功后，任务 project-unitest、project-pep8 和 project-pyflakes 才会被执行。这在很大程度上可以避免运行一些无用的任务。

其中 noop 是 Zuul 内置的一个特殊任务，它永远立即返回 SUCCESS。当一个项目没有对应的任务，但所有的变更都希望按 pipeline 进行处理时，非常有用。

下面以 openstack/nova 项目为例，代码清单 5-14 只是列出了 nova 项目的部分任务。

代码清单 5-14　Openstack/nova 配置示例

```
- name: openstack/nova
    check:
        - gate-nova-tox-functional-ubuntu-trusty
        - gate-nova-tox-functional-ubuntu-xenial
        - gate-nova-tox-functional-py35-ubuntu-xenial
        - gate-tempest-dsvm-cells-ubuntu-trusty
        - gate-tempest-dsvm-cells-ubuntu-xenial
    # 一般来说，gate pipeline 中的任务是 check pipeline 的一个子集，可以相同
    gate:
        - gate-nova-tox-functional-ubuntu-trusty
        - gate-nova-tox-functional-ubuntu-xenial
        - gate-nova-tox-functional-py35-ubuntu-xenial
        - gate-tempest-dsvm-cells-ubuntu-trusty
        - gate-tempest-dsvm-cells-ubuntu-xenial
    post:
        - nova-coverage-ubuntu-trusty
        - nova-coverage-ubuntu-xenial
```

5.6.4 Project Templates

当有许多类似的项目，比如插件类项目，希望使用相同的 pipeline 和配置时，可以采用本节定义的项目模板（project templates）。项目模板定义了 pipeline 以及运行在该 pipeline 中的任务列表。对于这类项目，直接引用该项目模板即可，比较容易进行维护和更新。

下面看一个项目模板定义的例子，在代码清单 5-15 定义了一个名为 plugin-triggering 的模板，它有一个参数 jobprefix。

代码清单 5-15　Project 模板定义示例

```
project-templates:
    # 项目模板名称
    - name: plugin-triggering
      check:
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
            # 可以有参数，如 '{jobprefix}'，也可以是没有参数的任务，如 'anteater'
            - '{jobprefix}-merge':
                - '{jobprefix}-pep8'
                - '{jobprefix}-pyflakes'
                - 'anteater'
        gate:
            - '{jobprefix}-merge':
                - '{jobprefix}-unittest'
                - '{jobprefix}-pep8'
                - '{jobprefix}-pyflakes'
            - 'anteater'
```

在需要使用的项目中，直接引用即可，如代码清单 5-16 所示，项目 plugin/foobar 使用了 plugin-triggering 模板，并且把参数 jobprefix 的值设置为 plugin-foobar。

代码清单 5-16　Project 模板引用示例

```
projects:
    - name: plugin/foobar
        template:
            - name: plugin-triggering
                # 传递参数 jobprefix 的值为 plugin-foobar
                jobprefix: plugin-foobar
```

一个模板可以有多个参数，名为 parameter 参数的值通过展开 {parameter} 宏字符串来使用。参数 name 自动提供并包括项目的名称，即项目名称（project name）中 / 字符后面的字符串。

一个项目也可以组合使用多个项目模板，所有这些项目模板中定义的任务都会被最终添加到该项目中。当然，除了可以使用项目模板外，独立的任务也可以添加进来一起使用。

代码清单 5-17　多 Project 模板引用示例

```
projects:
    - name: plugin/foobar
        template:
            - name: plugin-triggering
                jobprefix: plugin-foobar
            - name: plugin-extras
                jobprefix: plugin-foobar
        check:
            - foobar-extra-special-job
```

如代码清单 5-17 所示，项目 plugin/foobar 使用了两个模板：plugin-triggering 和 plugin-extras，以及一个名为 foobar-extra-special-job 的独立任务。

项目中这些任务出现的次序是：先是项目模板中的任务，然后是独立任务，来自于项目模板的任务与在项目模板中的次序完全一致。这个次序只会影响 Zuul 向 Gerrit 上报时，任

仅供非商业用途或交流学习使用

务出现的次序，没有其他作用，没有特殊设置的情况下，这些任务都是并行执行的。

需要注意的是，如果项目引用了多个项目模板，并且多个项目模板使用了相同的任务，或项目本身也直接指定了该任务，最后一个引用的模板或项目指定的任务具有最高优先级。

5.7 本章小结

总体而言，Zuul 为项目开发提供了一个有效的门控系统，有如下特点：

- ❑ 测试后合入，确保主线稳定。
- ❑ 并行执行，保障开发效率。
- ❑ 自动依赖，保证结果可靠。
- ❑ 自动回滚，尽可能准确、自动化、并行编排测试任务。
- ❑ 项目依赖，解决跨项目集成。
- ❑ 通用任务，提升项目整体基础设施维护的效率。

第 6 章 Chapter 6

资源管理系统（Nodepool）

本章先介绍 Nodepool 的原理和相关组件，然后结合实践讲解如何配置 Nodepool，由于 Nodepool 使用了镜像制作工具，而镜像制作在 OpenStack CI/CD 中较为关键，所以接下来会单独探讨下社区镜像制作工具及其大致原理。本章的读者需要对 OpenStack 的相关概念（如租户、FloatingIP、Region、AZ 等）有一定的了解。

6.1 Nodepool 简介

6.1.1 Nodepool 引入的背景

首先，在 OpenStackCI/CD 中，所有的变更在合入主分支前都要求进行单元测试、集成测试甚至性能测试，而使用 Zuul 作为门控系统后，同一个变更的同一个测试也可能需要多次集成测试，这些测试需要大量的 Slave 作为支撑，所以如何有效管理这些 Slave 是一个亟待解决的问题。

其次，由于测试用例的多样性，对 Slave 要求也各不相同，比如对运行在 Slave 上的操作系统有的要求 Ubuntu、有的要求 CentOS，或者运行操作系统的种类相同但版本不同，或者仅仅是硬件规格不一样（有的是 CPU 密集型的，要求 CPU 资源较多；有的是 IO 密集型的，对内存或磁盘有较高的要求）；更为普遍的情况是，硬件规格要求一样，运行的操作系统和版本也相同，但是不同的测试依赖的软件包可能不一样：容器测试一般需要 Docker，开发环境有的需要 Python2 有的需要 Python3 等。如何定制这些 Slave 以满足不同测试的需求是 OpenStack CI/CD 要解决的又一大问题。

最后，对于同一类测试，都希望运行环境尽可能一样，这意味着如果在一个 Slave 上多

次运行某个测试，需要在上一次测试运行完毕后恢复环境，以保证下次测试不受历史数据的影响，而环境的清理本身就不是一件容易的事，有时其处理逻辑的工作量甚至会超过正常业务处理的工作量，所以有一种极端的做法就是运行完一次测试，就重新装一次系统，这显然不是我们所期望的。

为了解决以上问题，OpenStack 基础设施团队研发了 Nodepool 系统，即节点资源动态管理系统，通过自动定制云环境中的虚机（VM）来满足各种测试对运行环境的要求，由于云环境中的 VM 易于管理，可以根据需要，每执行完一次测试就自动重新产生一个全新的 VM 供下次测试使用，真正做到 Slave 中的测试彼此间相互不影响。此外，Nodepool 还使用了镜像制作工具，来满足对运行环境的定制化需求。

6.1.2 Nodepool 的功能

Nodepool 在 OpenStack CI/CD 中的位置如图 6-1 所示。

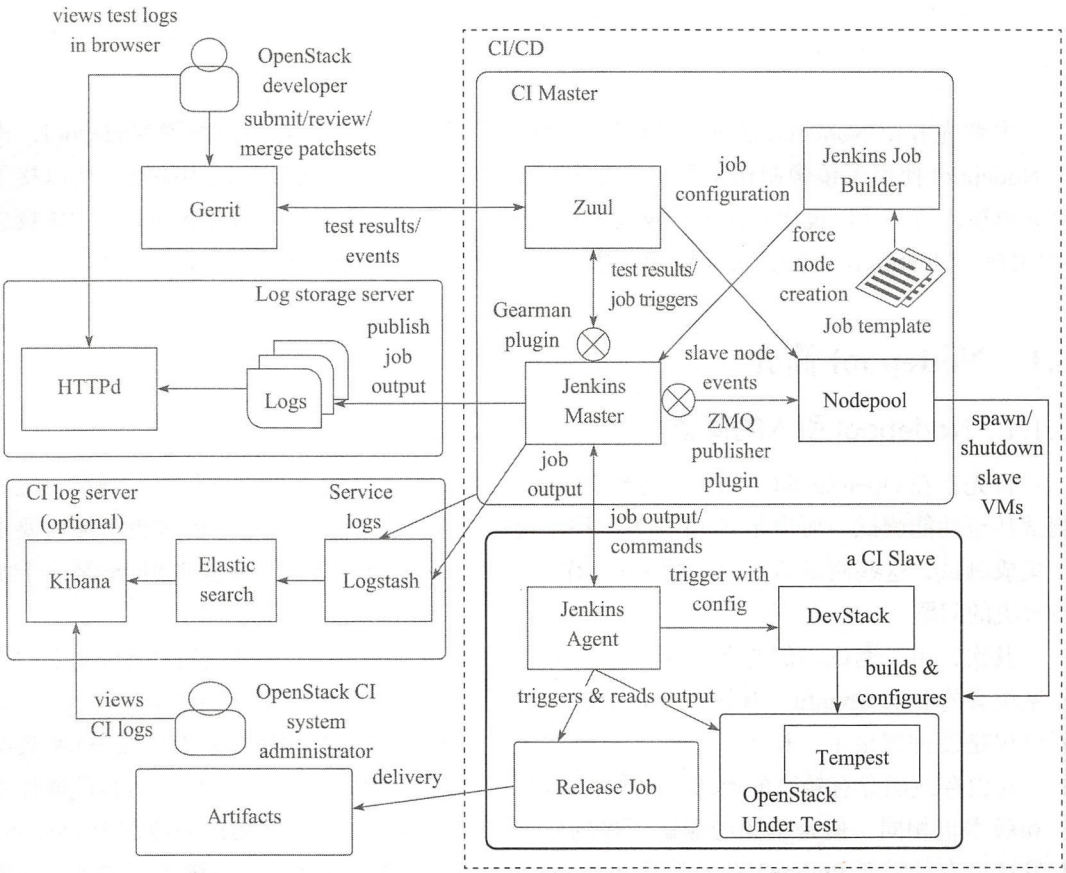


图 6-1 Nodepool 在 OpenStack CI/CD 中的位置

图中黑色框（加粗部分）里面即 Nodepool 包含的范畴，灰色（深色）部分为其他组件。

从 Slave 和 Jenkins 交互的角度，Slave 分为动态 Slave 和静态 Slave。静态 Slave 需要手动添加到 Jenkins 中，并一直存在于 Jenkins 中，除非手动从 Jenkins 中删除，静态 Slave 可以是物理机、虚机或容器；动态 Slave 一般是通过自动化工具管理其生命期的，即先通过工具创建 Slave，然后注册到 Jenkins 中，使用完毕后自动从 Jenkins 中删除，动态 Slave 目前多以虚机的形式存在。由于是人为控制，所以静态 Slave 可以在同一个生命期多次运行测试任务，这些测试可能相关，也可能不相关，而动态 Slave 是按自动化流程控制各生命节点的，一次生命期只运行一个测试任务（Slave executors 默认为 1），任务运行完 Slave 生命期就结束了，Nodepool 系统就是这样的 Slave 生命期自动化管理工具，其与 Jenkins 和 Slave 的关系如图 6-2 所示。

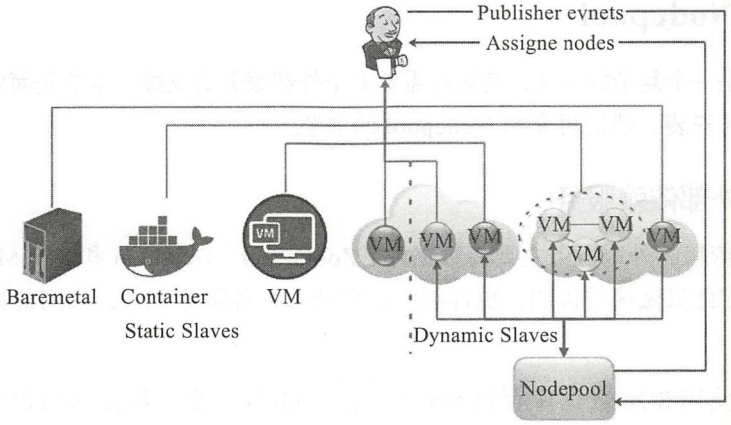


图 6-2　Nodepool 与 Jenkins 和 Slave 的关系

需要说明的是，Nodepool 管理的 Slave，不仅可以提供独立的 VM，还可以是一组 VM，组内 VM 彼此互通，但是在 Jenkins 看来就像只有一个 VM，这种情况是为满足一次测试需要多个 VM 协同工作的复杂场景而设计的。例如测试 kubernetes，一个 VM 用作 kubernetes master，其余作 kubernetes node。

如前所述，当前 Nodepool 仅管理 VM，所以 Nodepool 功能上都是围绕 VM 而来的，主要分两大部分，如下所述。

- VM 管理相关功能
  - 云配置管理：对接一个或多个云环境。
  - VM 生命期管理：按照配置和当前 Slave 使用情况在云环境中创建 VM，使用完毕后删除 VM。
  - Flavor 管理：从云中获取 Flavor 配置，并在创建 VM 时根据配置选择合适的 Flavor。
  - FloatingIP 管理：从云环境中为 VM 分配 FloatingIP，便于外部访问 VM，VM 删除时释放 FloatingIP。
  - Slave 注册注销管理：将云中分配到的 VM 通过 RESTful 接口注册到 Jenkins 中，

作为 Jenkins 的 Slave，使用完毕后从 Jenkins 中注销。

- 资源同步：同步云环境中的 VM 和供 Jenkins 使用的 VM，保持资源的一致性，防止 VM、FloatingIP 等资源的漏删。

❑ 镜像管理相关功能

- 制作镜像：根据 Nodepool 镜像配置需求定制镜像。
- 上传镜像：将定制好的镜像上传到云环境，供动态创建 VM 时使用。
- 清理镜像：镜像文件在 Nodepool 上有缓存，Nodepool 周期性清理本地过期不用的镜像文件，为新的镜像保留足够的空间。

6.2 安装 Nodepool

Nodepool 是一个复杂的系统，其运行需要多个外部服务的支撑，本节先简要说明 Nodepool 依赖外部服务的安装，然后再介绍 Nodepool 的安装。

6.2.1 准备外部依赖服务

Nodepool 依赖的外服服务主要有数据库、Zookeeper、Gearman 和 ZeroMQ 等[⊖]，这些服务的安装配置在此只做简单说明，更详细的过程请参见各服务对应的官网资料。

1. 数据库

Nodepool 使用 SQL 数据库存储 Slave 节点、Job 等信息。数据库可以选择 MySQL 或 Postgre-SQL，配置为 InnoDB 存储引擎的 MySQL 已经与 Nodepool 进行过较充分的测试，所以推荐使用；由于需要接收来自于 Nodepool 的高并发请求，不推荐使用 SQLite。如非特殊说明，本文后面数据库指的都是 MySQL。

由于每创建或者删除一个 Slave，都会新建一个数据库的连接，所以数据库连接数至少要大于期望使用 Slave 个数的两倍，下面是数据库连接数参数的配置示例，如代码清单 6-1 所示。

代码清单 6-1 设置 MySQL 的最大连接数

```
mysql> show global variables like "%max_connection%";
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| max_connections | 100   |
+-----------------+-------+
1 row in set (0.00 sec)

mysql> set global max_connections=200;
```

⊖ https://docs.openstack.org/infra/nodepool/installation.html。

```
mysql> show global variables like "%max_connection%";

+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| max_connections | 200   |
+-----------------+-------+
1 row in set (0.00 sec)
```

用户需要手动创建数据库，Nodepool 连上对应数据库后会自动创建需要的表信息，如代码清单 6-2所示。

代码清单 6-2　创建并给本地用户授权

```
CREATE USER 'nodepool'@'localhost' IDENTIFIED BY '<password>';
CREATE DATABASE nodepool;
GRANT ALL ON nodepool.* TO 'nodepool'@'localhost';
```

如果将数据库安装在独立的节点上，需要允许数据库远程访问，如代码清单 6-3 所示。

代码清单 6-3　授权给远程用户访问

```
CREATE USER 'nodepool'@'%' IDENTIFIED BY '<password>';
GRANT ALL ON nodepool.* TO 'nodepool'@'%';
```

Nodepool 连接上数据库后，创建数据库的结果如代码清单 6-4 所示。

代码清单 6-4　Nodepool 数据库表

```
mysql> show tables;
+--------------------+
| Tables_in_nodepool |
+--------------------+
| job                |
| node               |
| subnode            |
+--------------------+
```

2. Zookeeper

Zookeeper 在 Nodepool 中用来存储镜像信息。单实例 Zookeeper 可以与 Nodepool 部署在一起，如果 Nodepool 需要管理的镜像比较多，此时 Nodepool 本身消耗的资源会较大，为了避免 Nodepool 和 Zookeeper 相互影响，应该将 Zookeeper 配置成独立的节点或集群。如果选择集群，一般包含 3 个 Zookeeper 节点就足够了。

Zookeeper 的安装及配置请参见第 9 章。对于 Nodepool，只需要配置 Zookeeper 的服务器地址，Nodepool 连接上 Zookeeper 后会自动创建需要的 Znode 节点信息，如代码清单 6-5 所示。

代码清单 6-5 Zookeeper 中的 Nodepool 部分结构

```
[zk: localhost:2181(CONNECTED) 0] ls /
[nodepool, zookeeper]
[zk: localhost:2181(CONNECTED) 1] ls /nodepool
[images]
[zk: localhost:2181(CONNECTED) 2] ls /nodepool/images
[ubuntu-trusty-large, ubuntu-trusty]
[zk: localhost:2181(CONNECTED) 3] ls /nodepool/images/ubuntu-trusty
[builds]
...
```

注解 ubuntu-trusty-large 和 ubuntu-trusty 是作者环境中的配置，实际会根据 Nodepool 中 label 的配置不同而有所差异。

3. ZeroMQ

Nodepool 通过 ZeroMQ 从 Jenkins 接收 Slave 的状态通知信息。Jenkins 服务器通过插件的方式来支持 ZeroMQ，Jenkins 插件的具体安装请参见第 4 章中的描述。Nodepool 需要配置 ZeroMQ 的地址，并确保 Nodepool 可以正确连接到 Jenkins 服务器中指定的端口。

4. Gearman

Nodepool 在创建 Slave 时，可能需要从 Zuul 和 Jenkins 中获取 Slave 的当前使用情况，这些是通过 Gearman 进行的，Gearman 可以使用 Zuul 自带的服务（参见第 5 章），也可以使用独立的 Gearman 服务（参见第 9 章）。需要说明的是，Nodepool 也可以在不使用 Gearman 的情况下工作，此时工作在“节点替代模式”，即 Nodepool 只有在删除一个 Slave 时才创建一个新的 Slave，而不会提前创建 Slave。不过，一旦配置了 Gearman，请确保 Nodepool 和 Gearman 是可以通信的，否则 Nodepool 不能正常工作。

可以通过 gearadmin --status 或 gearadmin --workers 查看 Nodepool 和 Gearman 工作是否正常，如代码清单 6-6 所示。

代码清单 6-6 查看 gearman 状态

```
root@common:~# gearadmin --status
build:zaas-docbuild-rst:ubuntu-trusty  0      0      13
build:openzero-blackduck-scan-query:ubuntu-trusty 0   0        13
build:knitter_exec_build:K8S-CI_build  0      0      1
build:salt-formula-kubernetes-real-update:ubuntu-trusty-large  0    0      8
```

注解 以上信息的含义请参见第 9 章中 Gearman 的相关介绍。

5. Statsd 和 Graphite

该配置是可选的，Statsd 负责收集并合并测量值信息，然后将这些处理后的测量信息

传给 Graphite，后者以时间序列为依据存储数据，并绘制图表。如果已经有一个 Statsd&-Graphite 系统，Nodepool 也可以直接与其进行集成，用于采集 Slave 和镜像使用情况。配置时，通过设置环境变量 STATSD_HOST 为 Statsd 的域名，设置 STATSD_PORT 为 Statsd 的端口（默认为 8125）。表 6-1 所示的是 Nodepool 中部分测量信息。

表 6-1　Nodepool 中的部分测量信息

测量项	说明
nodepool.nodes.STATE	节点状态（如 ready、deleting、used 等）
nodepool.target.TARGET.nodes.STATE	Jenkins 下的节点状态
nodepool.label.LABEL.nodes.STATE	标签下的节点状态
nodepool.provider.PROVIDER.nodes.STATE	云环境下的节点状态
nodepool.launch.provider.PROVIDER.subkey	云环境下的附加状态信息（如 error 信息）
nodepool.launch.image.IMAGE.subkey	镜像中的附加状态信息
nodepool.launch.subkey	节点启动的附加状态信息

6.2.2　安装 Nodepool

安装和配置好 Nodepool 的外部依赖后，就可以安装 Nodepool 服务了，Nodepool 需要 Python2.7 或更新版本。

使用 git 通过源码方式安装：

```
git clone https://github.com/openstack-infra/nodepool.git
cd nodepool && git checkout 0.5.0
pip install .
```

注解　当前 Nodepool 最新的版本只支持 Zuulv3，相关架构和之前版本不兼容，所以推荐使用源码方式安装（tag 0.5.0 之后的都是支持 Zuulv3 特性，所以需要检出 tag 0.5.0 或之前的版本）。

成功安装完 Nodepool 后，要使系统能正常运行，还需要对 Nodepool 做诸多的配置，为了理解各个配置的意义，我们需要先介绍 Nodepool 的相关设计原理。

6.3　Nodepool 的设计原理

Nodepool 系统框架及与其他系统的关系如图 6-3 所示。

Nodepool 包括 Nodepoold、Nodepool-builder 和 Nodepool Client 三个部分。

- Nodepoold：资源管理守护进程，简称资源管理，包括从配置的 OpenStack 环境中创建和删除 VM，以及将 VM 从 Jenkins 中注册注销等。

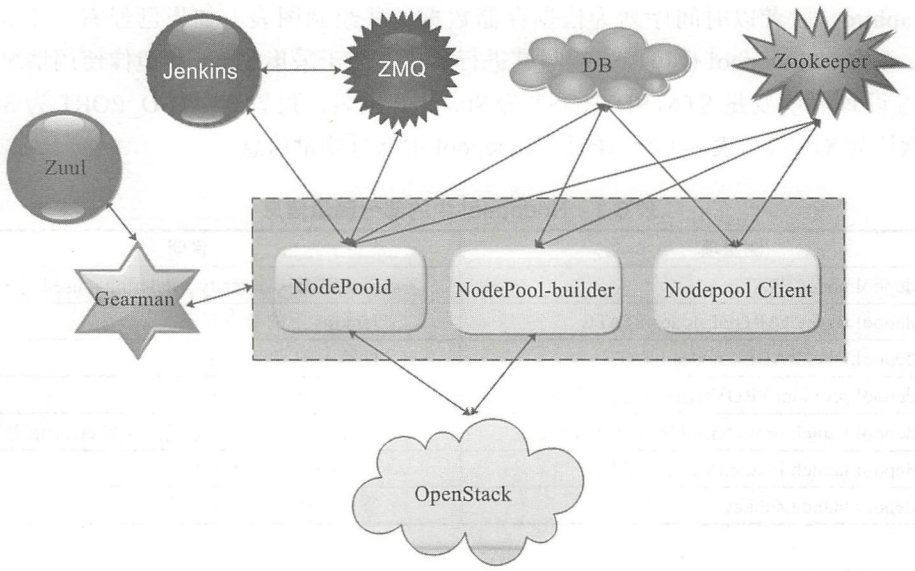


图 6-3 Nodepool 架构

- Nodepool-builder：镜像管理守护进程，简称镜像管理，负责镜像的制作和上传。
- Nodepool Client：Nodepool 客户端，对外提供多种 CLI，包括 VM、Job 和镜像的增、删、查等。

OpenStack CI/CD 中，与 Nodepool 有交互的主要外部组件有 Jenkins 和 Zuul，以及提供 Slave 运行环境的 OpenStack。除此之外，为了实现 Nodepool 上下文信息的持久化，需要数据库来保存已创建 VM 的信息。Nodepool-builder 是一个 CPU 和 IO 密集型的服务，且需要访问 internet，为了减少对 Nodepool 其他服务的影响，设计上将 Nodepoold 和 Nodepool-builder 进行解耦，两者可以分别部署到不同的服务器上，通过 Zookeeper 进行镜像配置信息交互。

Nodepool 内外部涉及的组件、模块及其功能和通信方式如表 6-2 所示。

表 6-2 Nodepool 内外部组件、模块及其通信方式

Nodepool 组件	方向	其他组件	通信方式	说明
Nodepoold	->	Jenkins	RESTful	注册 VM 注销 VM
Nodepoold	<-	Jenkins	ZeroMQ	广播 VM 状态
Nodepoold	<-	Jenkins	Gearman	获取 VM 使用状态
Nodepoold	<->	OpenStack	RESTful	创建 VM 删除 VM 查询 VMs 查询 Flavor 创建 FloatingIP 删除 FloatingIP
Nodepool-builder	->	OpenStack	RESTful	上传镜像

（续）

Nodepool 组件	方向	其他组件	通信方式	说明
Nodepoold	<–	Zuul	Gearman	获取 VM 请求状态
Nodepoold	<–>	Database	sql	增 / 删 / 改 / 查 VM 信息
Nodepoold	<–>	Nodepool Client	sql Zookeeper	查 / 删 VM 信息 增 / 查 / 删镜像信息
Nodepool-builder	–>	Nodepool Client	Zookeeper	查询镜像信息
Nodepool-builder	<–>	Client	Zookeeper	增 / 查 / 删镜像

下面分别详细介绍 Nodepool 的各个部分。

6.3.1　资源管理（Nodepoold）

资源管理也是 Nodepool 的主体，按照功能分为调度、节点启动器（NodeLauncher）、节点删除器（NodeDeleter）和周期性校验及检查等几个部分。

- 调度模块（NodePool）：动态获取云配置、Jenkins 配置、数据库配置、Gearman 配置、Zookeeper 配置等，并负责启动 NodeLauncher。
- NodeLauncher：从云环境中分配 VM，进行相关的配置（如分配 FloatingIP 和注入脚本等）后将 VM 注册给 Jenkins。
- NodeDeleter：接收 Jenkins 的通知，将 VM 从 Jenkins 中注销，并将其从对应的云环境中删除。
- 周期性校验和检查：周期性校验主要保证 VM 的有效性，Nodepool 会周期性检测到 VM 的网络连接，如果发现和 VM 网络不可达则认为 VM 不能对外提供服务，便将其从 Jenkins 和云环境中删除，并申请新的 VM 取代之。周期性检查主要负责清除历史遗留的 VM。

资源管理的主要工作流如图 6-4 所示。

过程简述如下：

1）以下情况只要任何一个出现，都会触发 Nodepool 创建 VM：

- 配置变更导致需要的 Slave 数增加。
- 通过 Nodepool Client 触发新建 VM。
- Jenkins 消耗了 Slave 而需要补充新的 Slave。

2）NodeLauncher 根据配置选择合适的云，使用最新的镜像创建 VM，并设置供外

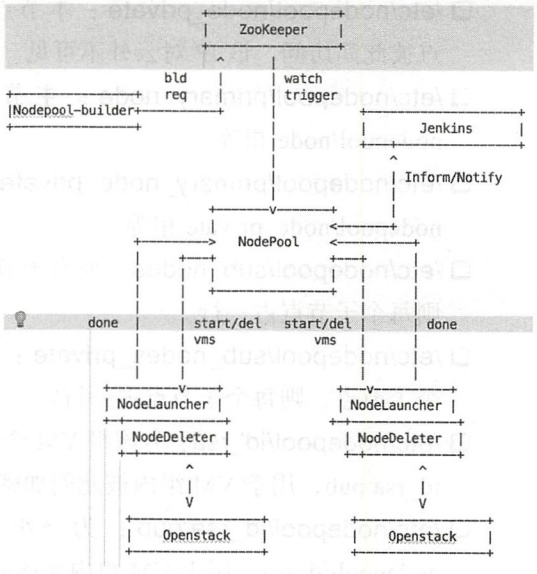


图 6-4　Nodepool 创建 VM 的主要流程

部访问的 Float-ingIP，然后注册给 Jenkins 用作 Slave。

3）Jenkins 运行完测试任务后，通过 ZeroMQ 广播 Slave 已使用完毕。

4）NodeDeleter 收到广播消息后调用 Jenkins 提供的 RESTful 接口从 Jenkins 中注销 Slave，并从云环境中删除对应的 VM 和 FloatingIP。

创建 VM 过程中涉及的细节：例如，如何选择云、如何选择 VM 的规格、需要产生多少 VM、如何命名 VM 等，将在 6.4 节讲解。

以上过程周而复始，始终保持 Jenkins 中的 Slave 数为配置中定义的数目。需要说明的是，在 VM 启动成功后，且在注册到 Jenkins 前，可以根据需要，自定义脚本对 VM 做一些初始化或测试等操作，只有这些操作成功后才会注册到 Jenkins，否则认为创建失败。

另外，对于多节点模式，即一次创建多个 VM，这些 VM 形成一个组为一次测试服务，则其中一个 VM 作为主节点，其余 VM 作为子节点，且只有主节点信息会注册到 Jenkins，子节点对 Jenkins 不可见。这种情况的一个应用场景是：对 Kubernetes 进行功能测试，主节点作为 Kubernetes master，子节点作为 Kubernetes node。为了方便运行在该组中的测试任务知道这是一个 VM 组（如：Kubernetes node 需要知道 Kubernetes master 的地址），节点创建时 Nodepool 会在该组所有节点中注入下列信息到固定的文件中。

- /etc/nodepool/role：本节点的角色，取值 primary 或者 sub，表明是主节点还是子节点，只有主节点会注册到 Jenkins 中。
- /etc/nodepool/node：本节点的 FloatingIP 地址，用于云外部访问本节点。
- /etc/nodepool/node_private：本节点的 fixedIP 地址，即租户内的 IP，用于组内节点彼此间访问，该 IP 对云外不可见。
- /etc/nodepool/primary_node：主节点的 FloatingIP 地址，主节点上该值和 /etc/nodepool/node 相等。
- /etc/nodepool/primary_node_private：主节点的 fixedIP 地址，主节点上该值和 /etc/nodepool/node_private 相等。
- /etc/nodepool/sub_nodes：所有子节点的 FloatingIP 地址，如果包含有多个子节点，则每个子节点占一行。
- /etc/nodepool/sub_nodes_private：所有子节点内部的 fixedIP 地址，如果包含有多个子节点，则每个子节点占一行。
- /etc/nodepool/id_rsa：为该组 VM 产生的 OpenSSH 私钥，对应公钥为 /etc/nodepool/id_rsa.pub，用于 VM 组内彼此间加密通信用。
- /etc/nodepool/id_rsa.pub：为该组 VM 产生的 OpenSSH 公钥，对应私钥为 /etc/nodepool/id_rsa，用于 VM 组内彼此间加密通信用。

- /etc/nodepool/provider：该组 VM 所在的云信息，主要包括如下几项：
  - NODEPOOL_PROVIDER：VM 云提供商的名字。
  - NODEPOOL_CLOUD：VM 所处云的名称。
  - NODEPOOL_REGION：VM 所处的 Region。
  - NODEPOOL_AZ：VM 所处的 available zone。
  - /etc/nodepool/uuid：VM 在云中的 ID，用于唯一标识该 VM。

基于以上信息，可以为 Slave 组织较为复杂的测试场景，满足特有的测试运行需求。

为了便于 Nodepool 对 VM 进行有效的管理，在创建时，Nodepool 会为 VM 添加一些元数据信息：

- groups：基于 Json 编码的列表，包含创建 VM 时使用的镜像名称，云提供商的名字等。
- nodepool：基于 Json 编码的字典，包含创建 VM 时使用的镜像名称，云提供商的名字，以及唯一标识 Nodepool 的 node_id 等。

下面是元数据在云中显示例子，如图 6-5 所示。

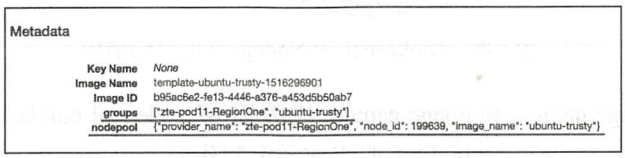


图 6-5　Nodepool 创建的 VM 元数据

6.3.2　镜像管理（Nodepool-builder）

镜像管理主要分为四个部分：镜像调度（NodePoolBuild）、镜像制作（BuildWorker）、镜像上传（UploadWorker），以及镜像清理（CleanupWorker）。顾名思义，BuildWorker 主要作用是根据配置动态生成创建 VM 需要的镜像文件，UploadWorker 将 BuildWorker 制作好的镜像上传到云环境中。NodePoolBuild 负责整个镜像的配置、制作和上传的调度管理；从命名上也可以看出，BuildWorker 和 UploadWorker 都可以配置为多实例的方式，以提高处理效率，默认 BuildWorker 运行 1 份，UploadWorker 运行 4 份。Nodepool 是一个独立的系统，在镜像上传到云环境后，都会在本地保留当前正在使用的镜像，另一方面，镜像制作自身也会留下大量的临时文件，如果不加以清理将会对存储带来较大的压力，这就是 CleanupWorker 引入的原因，CleanupWorker 会定期扫描镜像缓存目录和镜像存储目录，对超期文件进行清理。

前面介绍架构时已经提及，Nodepool-builder 和 Nodepool 是松耦合的，为了获得较高的效率，可以将 Nodepool-builder 部署到不同的服务器上，多实例条件下 Nodepool-builder 彼此独立，通过 Zookeeper 和 Nodepool 交换镜像信息，Nodepool 中 Zookeeper 的配置树结构如图 6-6 所示。

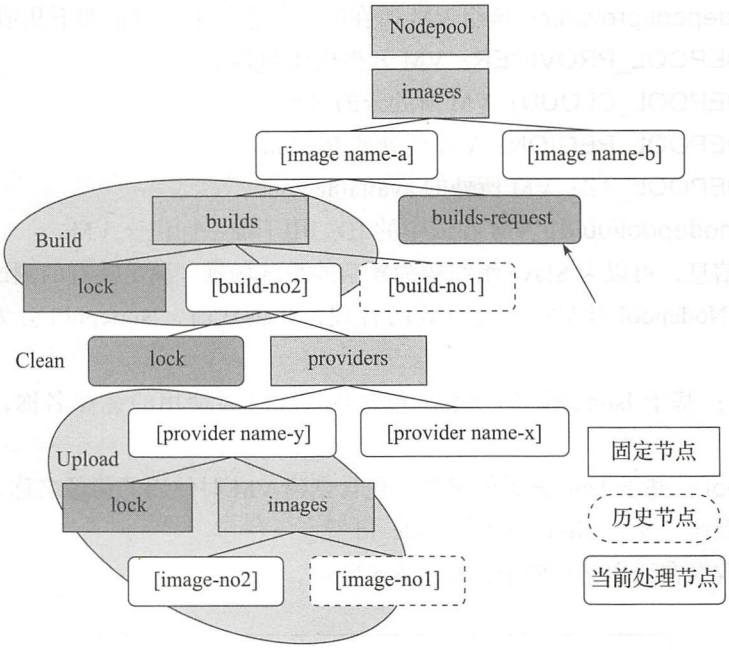


图 6-6　Zookeeper 中 Nodepool 的存储结构

图 6-6 中的 image name-a 和 image name-b 代表需要 Nodepool-builder 制作的镜像或者已经制作的镜像标签名称，request-build 表明当前需要制作 image name-a，当前制作的最新序号是 build-no2；对于制作的镜像 image name-a，会上传到两朵云中，分别是 provider name-x 和 provider name-y，其中 provider name-y 中对镜像 image name-a 进行了两次上传，上传序号分别是 image-no1 和 image-no2。对于制作的镜像，Nodepool-builder 都只保留最近成功的 2 个，而对于上传的镜像，只在每个云环境下保留 1 个成功的（如出现多个上传节点说明曾经上传失败过），其他的都会被 Cleanup-Worker 删除（前面已经说明，Nodepool 每次在新建 VM 时，总是使用对应云环境下最新的镜像），这意味着某个镜像在某个云环境下最多存在 1 个镜像文件。无论是制作镜像、上传镜像还是清理历史镜像，为保证 BuildWorker、UploadWorker 和 Cleanup-Worker 操作的有效性，都分别需要一把全局锁，其在 ZooKeeper 中的具体路径参见图 6-6 中的 lock 节点。

介绍完 ZooKeeper 中的节点结构后，我们看下镜像管理的主要工作流，如图 6-7 所示[⊖]。

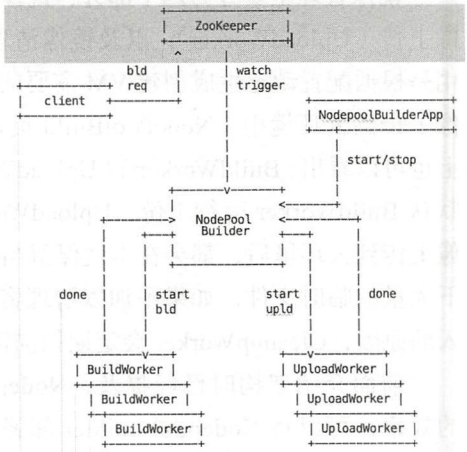


图 6-7　镜像制作主要流程

⊖ https://docs.openstack.org/infra/nodepool/devguide.html。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

过程简述如下：

1）有三种情况会触发镜像制作：镜像配置发生变更、通过客户端触发，或者镜像超过了生存期；此时会在 Zookeeper 树结构中对应的镜像标签下创建 builers-request 节点，表明该镜像需要制作。

2）处于守护状态的某个 BuildWorker 读取 builers-request，会首先加上制作锁，并生成新的制作序号，在对应 label 的 builds 节点下创建制作新节点如 build-no2。在此过程中如果其他 BuildWorker 扫描到 builers-request 的存在，会因为得不到锁而不会导致镜像重复制作。

3）得到锁的 BuildWorker 读取镜像的具体配置（主要是制作镜像用的 Elements 和镜像类型等），调用 disk-image-builder 工具生成镜像，成功后将节点状态设为 ready，释放制作锁，并删除 builers-request 节点。

4）同样处于守护状态的某个 UploadWorker 检测到有新的镜像制作完毕后，会扫描 Zookeeper 中需要该镜像的云所对应节点（如 provider name-y）下是否包含该镜像的信息，如果没有则会获取上传锁，在云节点的 images 下创建新的上传节点（如 image-no2），然后将镜像上传到对应云环境下，成功后将新创建节点状态设为 ready，最后释放上传锁。

在创建制作镜像节点 build-no2 和上传镜像节点 image-no2 时，BuildWorker 或 Upload-Worker 都会获取当前已经存在的节点序号，保证每次创建的序号都是最新的。为了便于管理和区别于云中的其他镜像，Nodepool-builder 对上传到云中的镜像也设有单独的元属性，主要是制作序号和上传序号，如图 6-8 所示的是镜像属性在云中的显示示例。

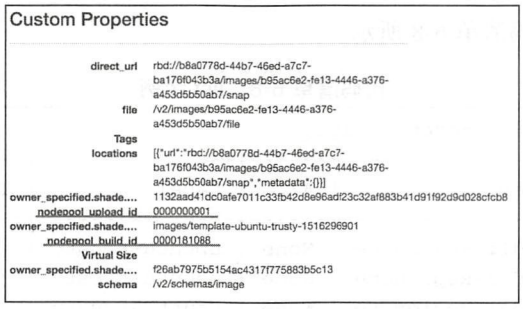


图 6-8　Nodepool 镜像的属性

6.3.3　客户端（Nodepool Client）

客户端是一个 CLI 工具，为用户提供了一个维护和调测 Nodepool 的手段。CLI 所有的子命令按功能分为配置相关、VM 相关、镜像相关和 Job 相关等几个部分。客户端设计上也采取了松耦合的方式，其中和 VM 相关的子命令操作和 Job 相关的操作是通过数据库和 Nodepoold 交互处理的，而和镜像相关的操作是通过 Zookeeper 和 Nodepool-builder 一起协作处理的[⊖]。

⊖ https://docs.openstack.org/infra/nodepool/operation.html。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

1. 配置相关操作命令

和配置相关的只有一个命令，即 config-validate，该命令校验当前配置是否正确，如果校验失败会给出错误信息。一般在修改配置后，都可以通过该命令进行预检查。该命令用法如代码清单 6-7 所示。

代码清单 6-7　config-validate 示例

```
nodepool@nodepool:~$ nodepool config-validate -c /home/nodepool/nodepool.yaml
2018-03-31 17:43:57,149 INFO nodepool.cmd.config_validator: validating /home/nodepool/
→nodepool.yaml
2018-03-31 17:43:57,178 INFO nodepool.cmd.nodepoolcmd: Configuation validation
→Vcomplete
```

其中 -c 指明了需要校验的配置文件，这是 Nodepool 客户端通用的参数，类似的还有：

- -s：安全配置相关参数，如云连接参数、Jenkins 连接参数及数据库连接参数等，默认为 /etc/nodepool/secure.conf。
- -l：log 配置文件，如果不配置，默认值输出 INFO 级别及 INOF 以上级别的信息，格式为“时间级别模块名日志信息”。
- --debug：是否打印调试信息，默认为 False。

2. VM 相关操作命令

（1）list

列出当前 Nodepool 创建的所有 VM 和状态，以及其对应在云环境和 Jenkins 中的部分信息，该命令用法如代码清单 6-8 所示。

代码清单 6-8　list 示例

```
nodepool@nodepool:~$ nodepool list
+--------+---------------------+------+---------------+----------+---------+
| ID     | Provider            | AZ   | Label         | Target   | Manager |
+--------+---------------------+------+---------------+----------+---------+
| 199923 | zte-pod11-RegionOne | None | ubuntu-trusty | jenkins1 | None    |
| 199924 | zte-pod13-RegionOne | None | ubuntu-trusty | jenkins1 | None    |
| 199925 | zte-pod11-RegionOne | None | ubuntu-trusty | jenkins1 | None    |
| 199926 | zte-pod13-RegionOne | None | ubuntu-trusty | jenkins1 | None    |
+--------+---------------------+------+---------------+----------+---------+

+------------------------------+--------------------------------+------------------+
| Hostname                                                      | NodeName
                  |
+------------------------------+--------------------------------+------------------+
| jenkins1-ubuntu-trusty-zte-pod11-RegionOne-199923 | jenkins1-ubuntu-trusty-zte-
→pod11-RegionOne-199923 |
| jenkins1-ubuntu-trusty-zte-pod11-RegionOne-199924 | jenkins1-ubuntu-trusty-zte-
→pod13-RegionOne-199924 |
| jenkins1-ubuntu-trusty-zte-pod11-RegionOne-199925 | jenkins1-ubuntu-trusty-zte-
→pod11-RegionOne-199925 |
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
    | jenkins1-ubuntu-trusty-zte-pod11-RegionOne-199926 | jenkins1-ubuntu-trusty-zte-
→pod13-RegionOne-199926 |
    +-----------------------------+-------------------------+------------------+
    +--------------------------------------+--------------+-------+------------+---------+
    | Server ID | IP | State | Age | Comment |
    +--------------------------------------+--------------+-------+------------+---------+
    | 1bbd16db-e41b-448f-a8cd-6f94d3f2aa0d | 172.110.0.137 | used | 00:01:48:48 | None |
    | 23728c3d-b5f7-44e6-95c0-744d3b43b5b4 | 172.110.0.157 | used | 00:01:48:45 | None |
    | b069e5f0-5917-4f4e-85c5-4805c3158229 | 172.110.0.145 | ready | 00:04:40:50 | None |
    | 16ea773a-c970-4fe6-ac9b-d61f3e983680 | 172.110.0.143 | ready | 00:04:41:35 | None |
    +--------------------------------------+--------------+-------+------------+---------+
```

- ID：VM 在 Nodepool 中的唯一标识，由 Nodepool 自动产生。
- Provider：VM 所处的云环境名称，命名方式为：<provider>-<region>，provider 和 region 来自于配置文件中 providers 下的 name 和 region-name。
- AZ：VM 所处的 Availability Zone，取值来自于配置文件中 providers 下的 availability-zones。
- Hostname：VM 在 Jenkins 中的名称，取值来自于配置文件中 targets 下的 hostname。
- Server ID：该 VM 在云环境中的唯一标识，该值由云环境自动产生。
- Label：VM 可以运行 job 的标签名，取值来自于配置文件中 labels 下的 name。
- Traget：VM 注册到了哪个 Jenkins，取值来自于配置文件中 targets 下的 name。
- NodeName：VM 注册到 Jenkins 中的名称，取值来自于配置文件中 targets 下的 hostname。
- IP：云环境分配给 VM 的 IP，显示浮动 IP，如果没有浮动 IP 则显示 Fixed IP。
- State：描述了当前 VM 在 Nodepool 中的状态，其中 used 表明该 VM 已经被 Jenkins 使用完毕，即将被删除，ready 表明可以运行 job，其他的状态还有：delete（VM 需要删除或正在被删除）、hold（VM 不会被自动删除）、building（VM 正在启动过程中）和 test（VM 已经启动成功，但仍处在测试状态，测试成功后会被设置为 readey）；
- Age：显示了当前 VM 从创建到当前所经历的时间。

（2）hold

手动让节点处于保持状态，而不会被自动删除，该命令在调试 Job 时比较有用，该命令用法如代码清单 6-9 所示。

代码清单 6-9　hold

```
nodepool@nodepool:~$ nodepool hold -h
usage: nodepool hold [-h] [--reason REASON] id

positional arguments:
    id                  node id

optional arguments:
    -h, --help          show this help message and exit
    --reason REASON     Optional reason this node is held
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

参数说明如下：

- id：指 VM 在 Nodepool 中的唯一标识，即 nodepool list 命令显示结果中 ID 一列的值。
- reason：用于说明 hold 的原因。

注解 处于 hold 状态的 VM 只能手动删除；hold 状态的 VM 也会占用云提供的 VM 数，为了便于维护，尽量提供 REASON 用以说明该 VM 为什么需要 hold。

（3）delete

删除 VM，即手动强制删除 VM，该命令用法如代码清单 6-10 所示。

代码清单 6-10 delete

```
nodepool@nodepool:~$ nodepool delete -h
usage: nodepool delete [-h] [--now] id

positional arguments:
  id          node id

optional arguments:
  -h, --help show this help message and exit
  --now       delete the node in the foreground
```

参数说明如下：

- id：指 VM 在 Nodepool 中的唯一标识，即 nodepool list 命令显示结果中 ID 一列的值。
- now：表明是否立即删除，如果不指定该参数，Nodepool 只设定 VM 状态为 delete，并在周期性检测任务中删除该 VM。

注解 不建议在 Jenkins 上删除 Nodepool 注册的 VM（Slave），因为 Jenkins 删除 VM 时不会通知 Nodepool，Nodepool 和 Jenkins 间也没有同步机制，所以这样操作的结果是在 Nodepool 中留下“孤立”的 VM，且孤立的 VM 依然会占用 Nodepool 提供的 VM 数，极端情况是 Nodepool 中已经存在大量处于 ready 状态的 VM，但 Jenkins 中其实无 VM 可用。而从 Nodepool 中删除 VM 时，不仅会删除云中的 VM，也会从 Jenkins 中将其注销。所以，删除 VM 应该由 Nodepool 自动发起，或通过 delete 命令手动发起。

3. 镜像相关操作命令

（1）image-list

列出镜像及其信息，包括制作信息、上传信息和在云中的使用情况。该命令用法如代码清单 6-11 所示。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

代码清单 6-11　image-list 示例

```
nodepool@nodepool:~$ nodepool image-list
+------------+------------+---------------------+---------------------+
| Build ID   | Upload ID  | Provider            | Image               |
+------------+------------+---------------------+---------------------+
| 0000000035 | 0000000001 | zte-pod11-RegionOne | ubuntu-trusty       |
| 0000000035 | 0000000001 | zte-pod13-RegionOne | ubuntu-trusty       |
| 0000000045 | 0000000002 | zte-pod11-RegionOne | ubuntu-trusty-large |
| 0000000045 | 0000000002 | zte-pod13-RegionOne | ubuntu-trusty-large |
+------------+------------+---------------------+---------------------+
+---------------------------------------+--------------------------------------+
| Provider Image Name                   | Provider Image ID                    |
+---------------------------------------+--------------------------------------+
| template-ubuntu-trusty-1525425224     | 4f158720-303e-4906-8c8f-ba8a47b65b28 |
| template-ubuntu-trusty-1524061372     | cc0077e3-d20b-45e8-9f57-bb27d9a2b4dd |
| template-ubuntu-trusty-large-1524052032 | 087e0e3b-0da8-4ccb-bb08-6ac2d536eceb |
| template-ubuntu-trusty-large-1524052040 | 898ef164-cd56-4a03-a973-924b0fb4f594 |
+---------------------------------------+--------------------------------------+
+-------+-------------+
| State | Age         |
+-------+-------------+
| ready | 04:17:02:39 |
| ready | 20:11:53:51 |
| ready | 20:14:29:25 |
| ready | 20:14:29:28 |
+-------+-------------+
```

- Build ID：镜像制作的唯一标识，从 1 开始顺序编号；每制作一次镜像，无论成功与否，该值加 1。
- Upload ID：某个镜像上传到指定云中的索引号，从 1 开始顺序编号，每上传一次，无论结果是否成功，都加 1。
- Provider：VM 所处的云环境名称，命名方式为：<provider>-<region>，provider 和 region 来自于配置文件中 providers 下的 name 和 region-name。
- Image：镜像名称，取值来自于配置文件中 diskiamges 下的 name。
- Provider Image Name：镜像在云环境中的名称，取值来自于配置文件中 providers 下的 template-hostname。
- Provider Image ID：镜像在云环境中的 ID，该值由云环境自动产生。
- State：描述了当前镜像在 Nodepool-builder 中的上传状态，其中 ready 表明该镜像已经成功上传到一个云环境，其他的状态还有：uploading（镜像正在上传）、deleting（镜像正在从云环境中删除）和 failed（镜像上传失败）。
- Age：显示了镜像从上传到云环境开始到当前所经历的时间。

（2）dib-image-list

列出制作的镜像及其信息，但不包含在云中的使用情况，该命令用法如代码清单 6-12 所示。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

代码清单 6-12 dib-image-list 示例

```
nodepool@nodepool:~$ nodepool dib-image-list
+--------------------------------+---------------------+-------------+
| ID                             |   Image             | Builder     |
+--------------------------------+---------------------+-------------+
| ubuntu-trusty-0000000035       | ubuntu-trusty       | nodepool.oz |
| ubuntu-trusty-0000000036       | ubuntu-trusty       | nodepool.oz |
| ubuntu-trusty-large-0000000045 | ubuntu-trusty-large | nodepool.oz |
| ubuntu-trusty-large-0000000046 | ubuntu-trusty-large | nodepool.oz |
+--------------------------------+---------------------+-------------+
+---------+----------+--------------+
| Formats | State    | Age          |
+---------+----------+--------------+
| qcow2   | ready    | 176:07:38:39 |
| qcow2   | building | 00:00:07:13  |
| qcow2   | ready    | 196:09:32:18 |
| qcow2   | building | 00:00:10:44  |
+---------+----------+--------------+
```

- ID：镜像文件的唯一标识，命名方式为：<Image>-<Build ID>，Image 和 Build ID 说明参见 6.3.3 一节中的说明。
- Builder：制作镜像的服务器名称。
- Formats：镜像格式，取值来自于配置文件中 providers 下的 image-type 和 diskiamges 下的 formats。
- State：描述了当前镜像在 Nodepool-builder 中的制作状态，其中 ready 表明该镜像已经制作成功，building 表明正在制作，其他的状态还有：deleting（镜像文件正在删除）和 failed（镜像制作失败）。
- Age：显示了镜像从制作开始到当前所经历的时间。

dib-image-list 中反映的是镜像制作的情况，包括正在制作的镜像，dib-image-list 中的镜像只有状态为 ready 后才能在 image-list 中查看到。需要注意的是，dib-image-list 中的一个镜像记录可能会在 image-list 中对应多条记录。此外，dib-image-list 中的每条记录，都会在本地磁盘上有对应的文件存在，制作成功的镜像如代码清单 6-13 所示。

代码清单 6-13 制作成功的镜像

```
nodepool@nodepool:/opt/nodepool_dib$ ll -h
total 2.0G
drwxrwxrwx 3 nodepool nodepool 4.0K Apr 8 11:20 ubuntu-trusty-0000000035.d/
-rw-r--r-- 1 nodepool nodepool 1012M Apr 8 11:22 ubuntu-trusty-0000000035.qcow2
-rw-rw-rw- 1 nodepool nodepool 89 Apr 8 11:22 ubuntu-trusty-0000000035.qcow2.md5
-rw-rw-rw- 1 nodepool nodepool 121 Apr 8 11:22 ubuntu-trusty-0000000035.qcow2.sha256
drwxrwxrwx 3 nodepool nodepool 4.0K Apr 28 11:43 ubuntu-trusty-large-0000000045.d/
-rw-r--r-- 1 nodepool nodepool 1012M Apr 28 11:46 ubuntu-trusty-large-0000000045.qcow2
-rw-rw-rw- 1 nodepool nodepool 89 Apr 28 11:46 ubuntu-trusty-large-0000000045.qcow2.md5
-rw-rw-rw- 1 nodepool nodepool 121 Apr 28 11:46 ubuntu-trusty-large-0000000045.qcow2.sha256
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

存放镜像的路径可以在配置文件中指定，示例中为 /opt/nodepool_dib，正在制作的镜像路径也可以配置，如 /opt/dib_tmp。

（3）image-build

手动触发制作新的镜像，旧的镜像不受影响，该命令用法如代码清单 6-14 所示。

代码清单 6-14　image-build

```
nodepool@nodepool:~$ nodepool image-build -h
usage: nodepool image-build [-h] image

positional arguments:
    image image name

optional arguments:
-h, --help show this help message and exit
```

其中 image 是待制作的镜像名称，取值来自于配置文件中 diskiamges 下的 name。

（4）image-delete

手动删除云环境中的镜像，该操作会触发新的镜像上传。该命令用法如代码清单 6-15 所示。

代码清单 6-15　image-delete

```
nodepool@nodepool:~$ nodepool image-delete -h
usage: nodepool image-delete [-h] --provider PROVIDER --image IMAGE
                             --upload-id UPLOAD_ID --build-id BUILD_ID

optional arguments:
    -h, --help                  show this help message and exit
    --provider PROVIDER         provider name
    --image IMAGE               image name
    --upload-id UPLOAD_ID
                                image upload id
    --build-id BUILD_ID         image build id
```

参数 provider、image、upload-id 和 build-id 的更多内容请参见 6.3.3 节中的说明。

（5）dib-image-delete

手动删除制作的镜像，该操作会触发新的镜像制作，该命令用法如代码清单 6-16 所示。

代码清单 6-16　dib-image-delete

```
nodepool@nodepool:~$ nodepool dib-image-delete -h
usage: nodepool dib-image-delete [-h] id

positional arguments:
    id dib image id

optional arguments:
-h, --help show this help message and exit
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

参数 id 的更多内容请参见 6.3.3 节中的说明。

（6）alien-list

此命令列出云环境中 Nodepool 可以查询到但又不在本 Nodepool 当前管理列表中的外部 VM。原理是：Nodepool 首先通过配置的云用户查询所有对接云上的 VM，如果 VM 在 Nodepool 数据库中不存在，则说明是外部 VM。这些外部 VM 可能是因为异常或配置错误导致云环境中的 VM 和 Nodepool 维护的 VM 不一致而留下的，此时只能手动从云中删除。该命令用法如代码清单 6-17 所示。

代码清单 6-17　alien-list 示例

```
nodepool@nodepool:~$ nodepool alien-list
+----------+----------+-----------+----+
| Provider | Hostname | Server ID | IP |
+----------+----------+-----------+----+
+----------+----------+-----------+----+
```

注解　基于以上检测的原理，如果一个系统中存在多个 Nodepool，都对接同一个云环境，建议为每个 Nodepool 配置独立的租户，这样彼此的 VM 相互不可见。

注解　alien-list 查询出来的 VM 与 Nodepool 泄漏的 VM 不同，泄漏的 VM 可以通过元数据确定是本 Nodepool 创建并出现了异常，所以可以通过自动清理流程删除，而 alien-list 查询出来的 VM 可能不是本 Nodepool 创建的，不能证明是否异常，所以不能贸然删除。

（7）alien-image-list

此命令可以列出云中 Nodepool 可以查询到但又不在本 Nodepool 当前管理列表中的外部镜像。原理是：Nodepool 首先通过配置的云用户查询所有对接云上的镜像，如果镜像在 Zookeeper 中不存在，则说明是外部镜像。这些外部镜像可能是因为异常或配置错误导致云环境中的镜像和 Nodepool 维护的镜像不一致而留下的，此时只能手动从云中删除。该命令用法如代码清单 6-18 所示。

代码清单 6-18　alien-image-list 示例

```
nodepool@nodepool:~$ nodepool alien-image-list
+---------------------+-------------------------------------------------+
| Provider            | Name                                            |
+---------------------+-------------------------------------------------+
| zte-pod13-RegionOne | ci-book-template-ubuntu-trusty-1515582892       |
| zte-pod13-RegionOne | ci-book-template-ubuntu-trusty-large-1515578873 |
| zte-pod11-RegionOne | ci-book-template-ubuntu-trusty-1515582892       |
| zte-pod11-RegionOne | ci-book-template-ubuntu-trusty-large-1515578873 |
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
+--------------------+--------------------------------------------------+

+--------------------------------------+
| Image ID                             |
+--------------------------------------+
| 8b81e5a5-616e-4e29-827b-e99c3cfff334 |
| 9947972a-ffb7-44b9-882b-d6a7ff00f677 |
| 94812d28-f491-480b-890a-2d3bb88636c8 |
| a4e864fc-f71f-42f2-a473-ec7ea2c7fa5f |
+--------------------------------------+
```

以上输出中，说明还有其他的 Nodepool 对接到云 zte-pod11-RegionOne 和 zte-pod13-RegionOne，因为当前 Nodepool 镜像名称在配置中都是以 template-ubuntu-* 命名的，不是 ci-book-*，所以本 Nodepool 将其列为外部的镜像并加以显示。

注解　dib-alien-list 查询出来的镜像与 Nodepool-builder 泄漏的镜像不同，泄漏的镜像可以确定是本 Nodepool-builder 创建并出现了异常，所以可以通过自动清理流程删除，而 dib-alien-list 查询出来的镜像可能不是本 Nodepool-builder 创建的，不能证明是否真正出现了异常，所以不能贸然删除。

4. Job 相关操作命令

有时 Job 会随机失败，而正常情况下，无论失败还是成功，用于 Slave 的 VM 都会在 Job 运行完毕后自动删除，此时如果想登录到 VM 上进行故障定位是不可能的，所以需要一种手段让那些运行指定 Job 失败的 VM 保持住，而不会被自动删除。下面这些命令就是为这个目的而设计的，可以先通过 job-create 指定需要调试的 Job，并指明保持多少个失败节点，以便定位问题，一旦问题处理完毕，则使用 job-delete 删除刚才创建的 Job，job-list 则用于查看当前 Job 的信息。

（1）job-create

创建测试 Job，该命令用法如代码清单 6-19 所示。

代码清单 6-19　job-create

```
nodepool@nodepool:~$ nodepool job-create -h
usage: nodepool job-create [-h] [--hold-on-failure HOLD_ON_FAILURE] name

positional arguments:
    name                  job name

optional arguments:
    -h, --help            show this help message and exit
    --hold-on-failure HOLD_ON_FAILURE
                          number of nodes to hold when this job fails
```

参数说明如下：

- Name：Job 的名称，所取值必须是和 Nodepool 对接的 Jenkins 上的某个 Job 名。
- hold-on-failure：当 Job 出现失败后，有多少个 VM 需要处于 hold 状态。

如果通过 job-create 创建了 Job，那么在该 Job 运行失败后，Nodepool 会自动将 Job 所处的前 hold-on-failure 个 VM 置于 hold 状态，而不会被自动删除。因为 hold-on-failure 次失败都需要从 Jenkins 触发，所以一般 hold-on-failure 取值不要太大。

该命令的使用例子参见代码清单 6-20。

代码清单 6-20　job-create 示例

```
nodepool@nodepool:~$ nodepool job-create zaas-real-deploy --hold-on-failure 5
+----+------------------+-----------------+
| ID | Name             | Hold on Failure |
+----+------------------+-----------------+
| 1  | zaas-real-deploy | 5               |
+----+------------------+-----------------+
```

示例表明为 zaas-real-deploy 这个 Job 保持 5 个失败的 VM。

（2）job-list

列出当前测试 Job 及其对应的处于 hold 状态的 VM 数，如代码清单 6-21 所示。

代码清单 6-21　job-list 示例

```
nodepool@nodepool:~$ nodepool job-list
+----+------------------+-----------------+
| ID | Name             | Hold on Failure |
+----+------------------+-----------------+
| 1  | zaas-real-deploy | 2               |
+----+------------------+-----------------+
```

- ID：Job 的唯一标识，由系统自动产生。
- Name：Job 的名称。
- Hold on Failure：当 Job 出现失败时，当前有多少个 VM 需要处于 hold 状态。

（3）job-delete

删除测试 Job，该命令不会删除因为指定 Job 运行失败而设置为 hold 状态的 VM，只删除数据库的 Job 记录，该命令用法如代码清单 6-22 所示。

代码清单 6-22　job-delete

```
nodepool@nodepool:~$ nodepool job-delete -h
usage: nodepool job-delete [-h] id

positional arguments:
    id          job id
```

```
optional arguments:
    -h, --help    show this help message and exit
```

参数 id 的更多内容请参见 6.3.3 节中的说明。

注解　因为 job-delete 不会删除已处于 hold 状态的 VM，所以如果某个 VM 被自动设为 hold 状态，只能手动删除。

6.4　配置 Nodepool

Nodepool 的配置主要涉及四个文件 clouds.yaml、secure.yaml、/etc/nodepool/nodepool.yaml 和 /etc/nodepool/secure.conf，按照 Nodepool 对外的连接关系分为云配置、Jenkins 相关配置、镜像相关配置以及其他配置（如数据库配置、Zookeeper 配置、Gearman 配置、周期性任务配置等）等[⊖]。

6.4.1　云相关配置

当前 Nodepool 中用到的 Slave 都来自于云环境中的 VM，所以正确配置云环境是 Nodepool 中的重要内容，该部分的配置主要位于 clouds.yaml 和 /etc/nodepool/nodepool.yaml 中。总体来说，云相关配置包含两个方面的内容，一个是云连接参数（clouds）配置，另一个是云资源相关参数（providers）配置，前者描述了 Nodepool 如何连接到云环境，后者定义了 Nodepool 如何使用云环境。

1. 云连接参数（clouds）

在原理中介绍过，Nodepool 可以连接多个云环境，而云连接配置本身就比较复杂，在较早的 Nodepool 版本中，云连接参数和云资源相关参数是放在一个文件中的，后来为了云管理的方便，将云环境连接相关的大部分参数（如云认证信息、网络信息、域信息以及云环境提供服务的版本信息等）独立出来，并使用 os_client_config 组件进行管理，将云连接作为一个连接池，云资源相关参数配置中只需通过简单的引用即可，这样屏蔽了云连接相关的细节，增加了 Nodepool 管理云的灵活性。云连接参数放在文件 clouds.yaml 和 secure.yaml 中，使用时会以下面顺序搜索这些文件：

（1）当前目录

（2）~/.config/openstack

（3）/etc/openstack

主要参数说明如下：

- client：可选参数，云环境是否支持 ipv6，还是强制为 ipv4：

⊖ https://docs.openstack.org/infra/nodepool/configuration.html。

- force_ipv4：可选参数，取值为 true，表明强制支持 ipv4，如果为 false，表明不强制使用 ipv4，默认为 false。
- prefer_ipv6：可选参数，取值为 true，表明优先使用 ipv6，如果为 false，则系统自动将 force_ipv4 置为 true，默认为 false。

❑ proftle：可选参数，前面说过，Nodepool 通过使用 os_client_config 屏蔽了到云环境的连接细节，这样不同的厂商只要按照 os_client_config 的要求提供不同的云连接文件，厂家特有的云就可以被 Nodepool 使用，profile 参数就是用来对接私有云的，它是一个网络管理策略名称，对应一个同名的 json 文件，其中配置对接私有云用到的一系列参数，包括认证信息和 region 等。如下面代码清单 6-23 所示的是 internap 策略对应的文件内容。

代码清单 6-23　internap 云连接参数

```
{
    "name": "internap",
    "profile": {
        "auth": {
            "auth_url": "https://identity.api.cloud.iweb.com"
        },
        "regions": [
        "ams01",
        "da01",
        "nyj01",
        "sin01",
        "sjc01"
        ],
        "identity_api_version": "3",
        "floating_ip_source": "None"
    }
}
```

❑ floating_ip_source：可选参数，浮动 IP 使用的管理工具，默认为 neutron。
❑ api_timeout：可选参数，通过 API 访问云环境超时时间，如果不设置，则不做超时检查。
❑ service_type：可选参数，服务类型，如 compute，默认为空。
❑ service_name：可选参数，服务名称，如 compute，默认为空。
❑ image_format：可选参数，创建 VM 时使用的镜像格式。
❑ auth：可选参数，认证相关配置，描述连接 Openstack 服务时的认证参数，默认使用用户名和密码方式认证。
- auth_url：可选参数，认证服务器的 url 地址，如：

```
auth_url: http://172.20.0.11:5000/v2.0
```

- username：可选参数，访问云环境使用的用户名。

- password：可选参数，访问云环境使用的用户密码。
- project_name：可选参数，访问云环境使用的租户名。
  如果云环境使用安全连接，也可以通过 SSL 方式进行认证。
- key：可选参数，访问云环境所使用 SSL 的认证密钥路径，如：

```
key: /home/myhome/client-cert.key
```

- cert：可选参数，访问云环境所使用 SSL 的认证文件路径，如：

```
cert: /home/myhome/client-cert.crt
```

- cacert：可选参数，访问云环境所使用 CA 的文件路径，如：

```
cacert: /home/myhome/ca.crt
```

- verify：可选参数，SSL 认证是否启用，默认为 true。

❑ regions：可选参数，云环境中的 regions 配置列表，会被 providers 中的各 region-name 引用，每个 region 包含如下选项：

- name：必选参数，云环境的 region 名称。
- values：可选参数，region 下的相关配置，当前只有网络配置。
  - networks：必选参数，region 下网络相关配置列表，列表中每个配置包含如下信息：
    - name：必选参数，网络名称，取值必须是对应云环境下已存在的网络名。
    - routes_externally：可选参数，网络是否是外部网络，即分配浮动 IP 的网络，默认为 false。
    - nat_destination：可选参数，网络是否启用 NAT 功能，默认 false。
    - default_interface：可选参数，网络是否是 VM 默认使用的网络，多网络下有效，默认为 false，如果只有一个网络，则该网络强制为默认网络。

同一个 region 下，只能有一个网络配置 nat_destination 的值为 true，也只能有一个网络配置 default_interface 的值为 true，否则校验会出错。

如果 region 下的 networks 配置为列表，那么该 region 下的 VM 创建后会存在多个网口，每个网口分别连接不同的网络。

❑ identity_api_version：可选参数，认证服务即 keystone 的版本，默认为“2.0”。
❑ compute_api_version：可选参数，计算服务即 nova 的版本，默认为“2”。
❑ image_api_use_tasks：可选参数，镜像服务即 glance 的版本，默认为“2”。

❑ network_api_version：可选参数，网络服务的即 neutron 的版本，默认为“2”。

配置示例如代码清单 6-24 所示：

代码清单 6-24　云连接参数配置

```
client:
    force_ipv4: true
    prefer_ipv6: false
clouds:
    cloud1:
        floating_ip_source: 'neutron'
        auth:
            username: 'ciuser'
            password: 'ciuser'
            project_name: 'ciuser'
            auth_url: 'http://172.20.0.11:5000/v2.0'
        regions:
            - name: RegionOne
              values:
                networks:
                  - name: admin_floating_net
                    routes_externally: true
                  - name: net-ci
                    nat_destination: true
                    default_interface: true
cloud2:
    auth: .....
    key: /home/myhome/client-cert.key
    cert: /home/myhome/client-cert.crt
    cacert: /home/myhome/ca.crt
    verify: true
```

上面的例子配置了两个云环境，分别为 cloud1 和 cloud2，两个云环境都使用 IPV4，使用 neutron 管理浮动 IP，此外 cloud2 启用了 SSL 认证，cloud1 则没有启用。cloud1 定义了一个 region，即 RegionOne；该 region 下定义了两个网络，即 admin_floating_net 和 net-ci，其中 admin_floating_net 是分配浮动 IP 的外网，而 net-ci 是内部默认网络，在 net-ci 网络启用了 nat 功能。

为了提高云连接的安全性，可以将云连接配置 clouds.yaml 中的敏感信息（如用户、密码等）放在 secure.yaml 中，并对文件 secure.yaml 的权限加以特别限制，如只允许管理员查看和修改，使用时两个配置文件中的配置项会进行合并，如果配置项都存在则会使用 secure.yaml 中的配置，secure.yaml 中的配置示例如代码清单 6-25 所示。

代码清单 6-25　云连接配置参数 -secure

```
clouds:
    cloud1:
        auth:
            username: 'ciuser'
```

```
    password: 'ciuser'
    project_name: 'ciuser'
```

本示例中将代码清单 6-24 中云 cloud1 的认证信息（包括用户名、用户密码和租户等）放在安全文件 secure.yaml 中。

技巧　云连接参数配置文件 clouds.yaml 和 secure.yaml 的后缀也可以是 yml，如果配置是 json 格式，后缀还可以是 json。

2. 云资源相关参数（providers）

云资源相关参数描述了 Nodepool 如何使用云，换句话说就是云环境可以为 Nodepool 提供哪些资源，这些参数主要定义了云环境中可供 Nodepool 使用的 AZ 列表、最大 VM 数、Nodepool 创建 VM 可以使用的镜像以及 VM 启动相关参数等，该配置位于 /etc/nodepool/nodepool.yaml 文件中。

主要配置参数如下：

- cloud：可选参数，连接云环境名称，该值需要在 clouds.yaml 中存在。
- region-name：可选参数，提供给 Nodepool 的源环境中的 region 名称，如果不配置，则默认使用第一个 region。

技巧　cloud 和 region-name 唯一确定一个云环境，通过 cloud 和 region-name 可以得到云连接的相关配置。

- auth-url：可选参数，认证服务器的 url 地址，如：

```
auth-url: http://172.20.0.11:5000/v2.0
```

- username：可选参数，访问云环境使用的用户名。
- password：可选参数，访问云环境使用的用户密码。
- project-name：可选参数，访问云环境使用的租户名。

技巧　project-name 在有些版本中叫作 project-id、tenant-id 或 tenant-name。

注解　provider 使用云连接有两种方式：通过 cloud 引用 cloud.yaml 云连接参数或者直接在 nodepool.yaml 中定义连接参数（包括 auth_url、username、password 和 project_name 等），推荐使用 cloud 方式。

- name：必选参数，云环境的名称。
- max-servers：必选参数，云环境能为 Nodepool 提供的最大 VM 数目。
- image-type：可选参数，创建 VM 时使用的镜像格式，默认为 qcow2。

image-type 会初始化为 cloud.yaml 配置中的 image_format，所以 provider 中一般不用配置。

- availability-zones：可选参数，Nodepool 可以使用的 availability zone（AZ）列表；如果不设置该参数，创建 VM 时完全依赖于 nova 自己调度结果；如果设置了多个 AZ，Nodepool 创建 VM 时会随机选择一个，所以如果需要更精确地控制 Nodepool 使用的 AZ，需要在云环境中定义更详细的 AZ，然后再提供给 Nodepool 使用。
- boot-timeout：可选参数，VM 启动超时时间，即从 VM 处于 active 状态到 SSH 连接成功的最长等待时间，单位为秒；如果启动超时，Nodepool 认为创建 VM 失败，从而删除该 VM；该参数默认为 60 秒。
- launch-timeout：可选参数，VM 上电超时时间，即 VM 创建开始到 VM 处于 active 状态的最长等待时间，单位为秒；如果上电超时，Nodepool 认为创建 VM 失败，从而删除该 VM；该参数默认为 3600 秒。
- keypair：可选参数，创建 VM 使用的公钥，默认不使用。
- networks：可选参数，创建 VM 时使用网络名称列表，列表中每个配置包含如下信息。
- name：必选参数，网络名称，取值必须是对应云环境下已存在的网络名。
- ipv6-preferred：可选参数，取值为 true，Nodepool 向 Jenkins 注册 VM 时会尝试使用 IPV6 网络，如果失败或者配置为 false 会使用 IPV4 网络；该参数默认为 false。
- rate：可选参数，Nodepool 创建 VM 的最短间隔时间，单位为秒；如果有大量的 VM 需要同时创建，会给云环境带来性能上的压力，所以可以通过设置该参数减少 Nodepool 对云环境的冲击；该参数默认为 1.0。
- clean-floating-ips：可选参数，如果设置为 true，Nodepool 会周期性地尝试清理没有使用的浮动 IP，这样可以避免浮动 IP 泄漏，默认为 false。
- api-timeout：可选参数，通过 API 访问云环境超时时间，如果不设置则不做超时检查。
- service-type：可选参数，服务类型，如 compute，默认为空。
- service-name：可选参数，服务名称，如 compute，默认为空。

api-timeout、service-type 和 service-name 推荐在 clouds.yaml 中配置。

- template-hostname：可选参数，镜像在该云环境下的名称模板，如 {image.name}-{timestamp}-nodepool-template，创建 VM 时 image.name 和 timestamp 会被替换为具体的值，还有一个变量是 {provider}；该参数默认为 template-{image.name}-{timestamp}。
- images：必选参数，定义了 Nodepool 在云环境下创建 VM 时可以使用的镜像列表，列表中的每项都必须存在于镜像配置的“diskimages”中，其中：
  - name：必选参数，镜像的名称。
  - min-ram：必选参数，表明创建 VM 时需要的最小内存大小。
  - name-ftlter：可选参数，取值为字符串，用于描述选择 falvor 时的附加信息，如设置 performance 表明需要选择元数据包含 performance 的 flavor，如果有多个 flavor 满足条件，则随机选择一个。

技巧　通过同时设置 min-ram 和 name-ftlter 两个参数可以精确选择创建 VM 时使用的 flavor。

- pause：可选参数，表明是否使用该镜像，true 表示使用，false 表示不使用；默认为 false。
- username：可选参数，Nodepool 创建 VM 后，通过 SSH 使用该参数指定的用户名连接 VM；默认为 jenkins。
- private-key：可选参数，Nodepool 通过 SSH 连接 VM 所使用用户秘钥的绝对路径，默认为 /var/lib/jenkins/.ssh/id_rsa。
- conftg-drive：可选参数，Nodepool 创建 VM 时是否使用 config-drive，默认为 false。
- meta：可选参数，Nodepool 创建 VM 时，注入的元数据信息，为一系列 key-value 对，key 和 value 长度都不超过 255。

配置示例如代码清单 6-26 所示。

代码清单 6-26　云资源相关参数配置

```
providers:
    - name: provider1
    cloud: example
    region-name: 'region1'
    max-servers: 96
    rate: 1.0
    availability-zones:
        - az1
    boot-timeout: 120
    launch-timeout: 900
    template-hostname: 'template-{image.name}-{timestamp}'
    ipv6-preferred: False
    networks:
        - name: 'some-network-name'
```

非卖品，仅供非商业用途或交流学习使用


```
    images:
        - name: trusty
          min-ram: 8192
          name-filter: 'something to match'
          image-type: qcow2
          username: jenkins
          user-home: '/home/jenkins'
          private-key: /var/lib/jenkins/.ssh/id_rsa
    meta:
        key: value
        key2: value
    - name: precise
      min-ram: 8192
      username: jenkins
      user-home: '/home/jenkins'
      private-key: /var/lib/jenkins/.ssh/id_rsa
    - name: devstack-trusty
      min-ram: 30720
      username: jenkins
      private-key: /home/nodepool/.ssh/id_rsa
- name: provider2
    username: 'username'
    password: 'password'
    auth-url: 'http://auth.provider2.example.com/'
    project-name: 'project'
    service-type: 'compute'
    service-name: 'compute'
    region-name: 'region1'
    max-servers: 96
    rate: 1.0
    template-hostname: '{image.name}-{timestamp}-nodepool-template'
    images:
        - name: precise
          min-ram: 8192
          username: jenkins
          user-home: '/home/jenkins'
          private-key: /var/lib/jenkins/.ssh/id_rsa
          meta:
            key: value
            key2: value
```

示例中为 Nodepool 定义了两个云环境，分别是 provider1 和 provider2，其中 provider1 使用了 cloud+region-name 方式连接云，最大提供 96 个 VM，使用的 AZ 为 az1，网络名为 some-network-name（cloud 中定义的网络也会使用），同时为 Nodepool 提供了 3 种镜像，即 trusty（需要元数据注入）、precise 和 devstack-trusty，用三种镜像创建 VM 时需要的内存分别是 8GB、8GB 和 30GB；VM 创建成功后，Nodepool 登录 VM 的用户名都是 jenkins，密钥位于 /home/nodepool/.ssh/id_rsa。

provider2 直接使用了传统的方式，云连接信息直接显式定义在例子中，并指明了服务

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

类型为 compute，可以为 Nodepool 提供的最大 VM 数为 96。provider2 中只使用了 1 种镜像即 precise（需要元数据注入），使用该镜像创建 VM 需要的内存为 8GB，创建成功后连接 VM 的用户名为 jenkins，密钥位于 /var/lib/jenkins/.ssh/id_rsa 中。

6.4.2　Jenkins 相关配置

Nodepool 管理的 VM 最终都会作为 Jenkins 的 Slave，所以申请多少 VM、如何将这些 VM 注册到 Jenkins 中是 Nodepool 配置的又一个重要的内容，该部分配置位于 /etc/nodepool/secure.conf 和 /etc/nodepool/nodepool.yaml 中，按功能分为 Jenkins 连接参数配置、target 配置和 label 配置等。

1. Jenkins 连接参数

为了安全，Jenkins 的连接相关参数配置如用户名、apikey、url 以及 credentials 等放在 /etc/nodepool/secure.conf 中，配置要求如代码清单 6-27 所示。

代码清单 6-27　Jenkins 连接参数

```
[jenkins "{target_name}"]
url={url}
user={user}
apikey={apikey}
credentials={credentials}
```

- target_name：必选参数，连接的 Jenkins 名称，唯一标识一个 Jenkins 连接，如 jenkins1。
- url：必选参数，Jenkins 对外提供的 RESTful API 地址，如：http://jenkins1.example.com:8080。
- user：必选参数，Nodepool 通过 RESTful API 访问 Jenkins 的用户名，如 Jenkins。
- apikey：必选参数，Nodepool 通过 RESTful API 以 {user} 访问 Jenkins 的认证信息。

注解　apikey 可以通过访问 Jenkins webui 得到，路径为用户名→ configure → API Token，如图 6-9 所示。

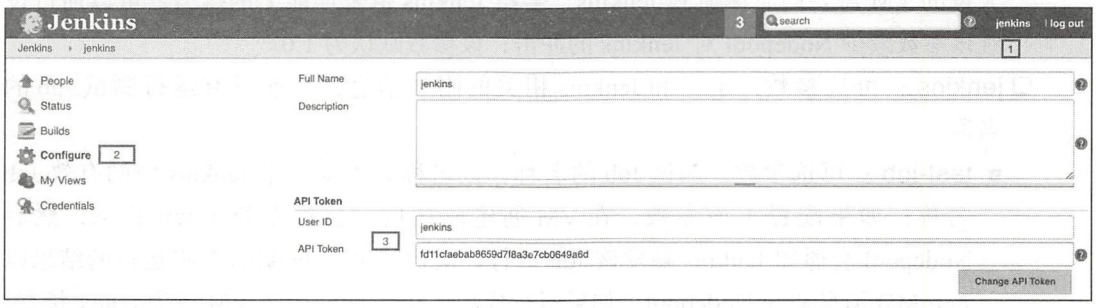


图 6-9　Jenkins API Key

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- credentials：必选参数，Jenkins 以 SSH 方式登录 Slave 执行 Job 任务时使用的认证信息，Jenkins 的 credentials 可以是用户名和密码，也可以是密钥等信息，具体可以参见第 4 章中的 credentials 相关描述。

Jenkins 连接参数配置示例如代码清单 6-28 所示。

代码清单 6-28　Jenkins 连接参数示例

```
[jenkins "jenkins1"]
url=http://jenkins:8080
user=jenkins
apikey=fd11cfaebab8659d7f8a3e7cb0649a6d
credentials=0e779e44-eb45-4266-864c-d729ab16c15a
```

该示例只为 Nodepool 定义了一个 Jenkins 服务器，名字为 jenkins1，对外 url 是 http://jenkins:8080，访问该服务器的用户名是 jenkins，认证信息 apikey 是 fd11cfaebab8659d7f8a3e7cb0649a6d，Jenkins 要访问 Nodepool 创建的 VM，需要使用 0e779e44-eb45-4266-864c-d729ab16c15a 指定的信息进行连接。

2. targets

targets 定义了 Nodepool 需要连接的 Jenkins，并规定了创建 VM 时的 hostname 等。target 配置位于文件 /etc/nodepool/nodepool.yaml 中。

主要参数如下：

- name：必选参数，Jenkins 名称，表明 VM 需要注册到哪个 Jenkins 中，取值于上文 secure.conf 的 target_name。
- hostname：可选参数，创建 VM 时的 hostname 模板，如 {label.name}-{provider.name}-{node_id}，变量在创建 VM 时会被具体值替代，还有一个可用的变量是 {timestamp}；该参数默认值为 {label.name}-{provider.name}-{node_id}。
- subnode-hostname：可选参数，创建子 VM 时的 hostname 模板，含义同上，该参数默认值为 {label.name}-{provider.name}-{node_id}-{subnode_id}。
- rate：可选参数，Nodepool 注册 VM 到 Jenkins 的最短间隔时间，单位为秒；如果有大量的 VM 需要同时注册到 Jenkins，会给 Jenkins 带来性能上的压力，可以通过设置该参数减少 Nodepool 对 Jenkins 的冲击；该参数默认为 1.0。
- jenkins：可选参数，定义和 Jenkins 相关的附加信息，目前只有运行测试 Job 的名称。
  - test-job：可选参数，测试 Job 的名称，该名称必须来自于 Jenkins 中的有效 Job 名称；如果配置了该参数，在 VM 创建成功后，会首先处于 test 状态，然后 Nodepool 会通知 Jenkins 触发该 Job 运行，运行结束后 Jenkins 会将运行的结果以 ZeroMQ 消息通知 Nodepool，如果运行成功，Nodepool 会将 VM 置为 ready 状态，如果失败，该 VM 将会被删除，如果一直没有运行结果通知给 Nodepool，该 VM

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

将会在周期性清理过程中被清除。

- assign-via-gearman：可选参数，是否立即通知 gearman 分配 Job 给新建的 VM，true 为立即通知，false 为不通知；默认为 false。

注解 hostname 既是 VM 在 Jenkins 中的显示名称，也是在云环境中的显示名称，subnode-hostname 代表的 VM 只会出现在云环境中，不会被注册到 Jenkins 中。

配置示例如代码清单 6-29 所示。

代码清单 6-29　Jenkins 连接相关参数配置

```
targets:
    - name: jenkins1
      hostname: '{label.name}-{provider.name}-{node_id}'
      subnode-hostname: '{label.name}-{provider.name}-{node_id}-{subnode_id}'
    - name: jenkins2
      hostname: '{label.name}-{provider.name}-{node_id}'
      jenkins:
        test-job: zaas-check
```

代码清单 6-29 说明了 Nodepool 创建的 VM 需要注册到两个 Jenkins 中，分别是 jenkins1 和 jenkins2，但同一时刻一个 VM 只能存在于一个 Jenkins 中；其中 jenkins1 中还定义了子节点的 hostname 模板，注册到 jenkins2 VM 需要通过 zaas-check 测试才能运行其他 Job。

3. labels

labels 描述了 Nodepool 需要为 Jenkins 创建多少类型（一个 label 表明一种类型）的 VM，每种类型的 VM 需要多少个，以及从哪些云中、使用什么镜像创建 VM，是否需要子节点等。Nodepool 会按照这些定义要求创建 VM，并注册到 Jenkins 中，这些 VM 在注册到 Jenkins 时会打上对应的标签类型，作为 Jenkins 运行 Job 时选择的依据。label 配置位于文件 /etc/nodepool/nodepool.yaml 中。

主要参数如下：

- name：必选参数，定义了标签名称，作为标识 VM 的类型。
- providers：必选参数，云名称列表，定义了指定类型的 VM 可以在哪些云中创建，这些云名称必须存在于 providers 定义的云列表中。
- image：必选参数，镜像名称，定义了创建指定类型的 VM 使用的镜像，该镜像名称必须存在于 providers 对应云的镜像列表中。
- min-ready：可选参数，该参数定义了 Nodepool 需要为 Jenkins 创建指定类型 Slave 的最小数目，默认为 2。
- subnodes：可选参数，该参数定义了是否需要为注册到 Jenkins 中的 VM 创建子节点，每个子节点也对应一个同种类型的 VM；如果定义具体值，Nodepool 则会为每

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

个注册到 Jenkins 的 VM 创建对应数目的子 VM，子 VM 不注册到 Jenkins 中，VM 和子 VM 之间可以相互通信；该参数默认为 0。

- ready-script：可选参数，该参数定义了在 VM 注册到 Jenkins 前需要通过的测试脚本，这些测试脚本是由 Nodepool 发起执行的。

注解 此处的 ready-script 和 target 中定义的 test-job 有所不同，前者是由 Nodepool 在 VM 中执行的本地脚本，脚本一般比较固定，且需要注入到 VM 镜像中，后者运行的是 Jenkins 上的 Job 所定义的脚本，较为灵活。两者的共同点都是由 Nodepool 主动发起的测试，只不过一个是直接执行，一个是间接执行。

配置示例如代码清单 6-30 所示。

代码清单 6-30 标签参数配置

```
labels:
    - name: my-precise
      image: precise
      min-ready: 2
      providers:
        - name: provider1
        - name: provider2
    - name: multi-precise
      image: precise
      subnodes: 2
      min-ready: 2
      ready-script: setup_multinode.sh
      providers:
        - name: provider1
```

代码清单 6-30 中定义了两种类型的节点，分别是 my-precise 和 multi-precise，前者需要 Nodepool 启动 2 个 VM 注册给 Jenkins 作为 Slave；后者需要 6 个 VM，并分成两组，每组一个主 VM，两个子 VM，且主 VM 会注册到 Jenkins 中。my-precise 由两个 provider 提供 VM，multi-precise 只有一个 provider 提供 VM。另外，multi-precise 还定义了 VM 必须通过的测试脚本 setup_multinode.sh，这个脚本是事先注入到 VM 中的。

6.4.3 镜像配置（diskimages）

镜像配置主要是定义制作镜像时用到的一些参数信息，重点是编译镜像需要的 Element，通过使用不同的 Element，就可以制作满足不同需求的 VM 镜像。此外，还可以配置镜像的基础版本、镜像的格式、镜像的老化时间，以及制作镜像时注入的一些环境变量等。镜像配置位于文件 /etc/nodepool/nodepool.yaml 中。

镜像配置主要参数如下：

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- name：必选参数，镜像名称，被云配置 provider 引用。
- pause：可选参数，该参数说明当前配置是否生效，true 为生效，false 为不生效，默认为 true。
- rebuild-age：可选参数，该参数定义镜像的老化时间，超过该时间镜像会重新制作，单位为秒，默认为 86400 秒。
- formats：可选参数，该参数定义了镜像格式列表，如 raw、qcow2 或 tar 等，具体取值参见 6.5 节中的说明；该参数默认为空。
- release：可选参数，定义了基础镜像的版本，这个配置需要结合 Element 基础 OS 配置，如取值 precise 时一定要选择 ubuntu 的基础 Element；默认为空。
- elements：可选参数，定义了镜像制作过程中用到的一系列脚本或文件，可以是标准的 Element 也可以自定义的 Element，如果包含自定义的 Element，需要指定环境变量 DIB_ELEMENT_PATH 或在配置文件中配置 elements-dir 参数；该参数默认为空。
- env-vars：可选参数，定义了制作镜像时用到的一些环境变量，可以是标准的环境变量，也可以是自定义变量（自定义 Element 中使用）。

注解　镜像配置中可以没有镜像类型，类型会取自于具体云环境下对应镜像的 image-type，如果同一镜像在不同的云配置下的格式不同，Nodepool 会为每个云配置生成一个镜像文件。

注解　虽然 elements 是可选参数，但是实际应用时都会配置为一定的值，以便将一些常用的包或配置做到镜像中，否则使用 Nodepool 功能上就会大打折扣。

镜像配置示例如代码清单 6-31 所示。

代码清单 6-31　镜像参数配置

```
diskimages:
    - name: precise
      pause: false
      rebuild-age: 86400
      elements:
        - ubuntu-minimal
        - vm
        - simple-init
        - openstack-repos
        - nodepool-base
        - cache-devstack
        - cache-bindep
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
            - growroot
            - infra-package-needs
        release: precise
        env-vars:
            TMPDIR: /opt/dib_tmp
            DIB_CHECKSUM: '1'
            DIB_IMAGE_CACHE: /opt/dib_cache
            DIB_APT_LOCAL_CACHE: '0'
            DIB_DISABLE_APT_CLEANUP: '1'
            FS_TYPE: ext3
    - name: xenial
        pause: true
        rebuild-age: 86400
        formats:
            - raw
            - tar
        elements:
            - ubuntu-minimal
            - vm
            - simple-init
            - openstack-repos
            - nodepool-base
            - cache-devstack
            - cache-bindep
            - growroot
            - infra-package-needs
        release: precise
        env-vars:
            TMPDIR: /opt/dib_tmp
            DIB_CHECKSUM: '1'
            DIB_IMAGE_CACHE: /opt/dib_cache
            DIB_APT_LOCAL_CACHE: '0'
            DIB_DISABLE_APT_CLEANUP: '1'
            FS_TYPE: ext3
```

代码清单 6-31 中定义了两个 ubuntu 镜像，分别是 precise 和 xenial，前者基于 Ubuntu precise 版本制作镜像，后者基于 Ubuntu xenial 版本制作镜像，两者定义了相同的 Element，说明需要预装相同的软件包。xenial 还指明了需要两种格式的镜像，分别是 raw 和 tar；由于 precise 中 pause 为 false，所以实际是不会制作 precise 镜像的。

关于镜像制作尤其是配置什么样的 Element，不是一件容易的事，需要对操作系统和镜像制作原理有一定的了解，6.5 节会讲解 Nodepool 中镜像制作的原理，可以作为镜像配置的入门参考。

6.4.4 其他配置

除了以上几个配置之外，Nodepool 还有其他一些配置，包括镜像文件路径的配置、Zookeeper 服务的配置、Gearman 服务的配置、周期性任务的配置等，这些配置都位于 /etc/

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

nodepool/nodepool.yaml 中。

1. 镜像目录

Nodepool 使用 diskimage-builder 和 glance 来制作镜像和上传镜像，镜像存储在本地，需要指明存储路径，在制作镜像的过程中，常常需要引用自定义的 Element，也需要为这些 Element 指定搜索路径，这些路径配置位于 /etc/nodepool/nodepool.yam 中。

- elements-dir：可选参数，自定义 Element 的存放绝对路径，默认为空。
- images-dir：必选参数，用来定义镜像存放的路径。

配置示例如下：

```
elements-dir: /etc/nodepool/elements
images-dir: /opt/nodepool_dib
```

2. 数据库配置

主要是数据库的连接信息，位于 /etc/nodepool/secure.conf 文件中，配置如下：

```
[database]
dburi={dburi}
```

其中 {dburi} 为数据库的配置，包括配置引擎和连接信息，例如：

```
dburi=mysql+pymysql://nodepool:root123@common/nodepool
```

3. 周期性任务

Nodepool 有多个周期性的任务，其中 cleanup 用于周期性清理过期的镜像和没有删除成功的 VM，check 用于周期性检查 VM 是否可用，具体通过 SSH 连接判断 VM 是否可用可达，如果不可达则认为该 VM 不可用。配置示例如代码清单 6-32 所示。

代码清单 6-32　周期性任务参数配置

```
cron:
    cleanup: '*/1 * * * *'
    check: '*/15 * * * *'
```

其中时间的格式符合 UNIX cron 风格[⊖]，但具体取值有所差别，由 * 分割的前五个字段含义分别是：

- 分取值：0 ～ 59
- 时取值：0 ～ 23
- 天取值：1 ～ 31
- 月取值：1 ～ 12
- 星期取值：0 ～ 6 或 mon、tue、wed、thu、fri、sat、sun（和 UNIX cron 不同）

与标准 UNIX cron 风格的另一个差别是，如果配置第六个参数，则会被解释为秒，取

⊖ https://docs.acquia.com/article/cron-time-string-format。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

值范围是 0 ～ 59。

4. ZeroMQ

定义了 Jenkins 提供的 ZeroMQ 服务端点（地址 + 端口）列表，Nodepool 通过该连接获取 Jenkins 关于节点的实时消息，对于使用完毕的节点，Nodepool 会启动删除节点流程，配置示例如代码清单 6-33 所示。

代码清单 6-33　ZeroMQ　连接参数配置

```
zmq-publishers:
- tcp://jenkins1.example.com:8888
- tcp://jenkins2.example.com:8888
```

5. Gearman-servers

Gearman 服务器列表，端口默认 4730，Nodepool 连接到 Gearman 服务器上以便实时获取需要新建节点的相关信息，其中等待运行的 Job 信息来自于 Zuul，正在运行的 Job 信息来自于 Jenkins，配置示例如代码清单 6-34 所示。

代码清单 6-34　Gearman 参数配置

```
gearman-servers:
    - host: zuul.example.com
      port: 4730
```

6. zookeeper-servers

Zookeeper 服务列表，用于 Nodepool 内部各组件之间交换镜像相关信息，配置示例如代码清单 6-35 所示。

代码清单 6-35　Zookeeper 参数配置

```
zookeeper-servers:
    - host: zk1.example.com
      port: 2181
      chroot: /nodepool
```

其中 port 是可选参数，默认 2181；chroot 也是可选参数，用于定义 Zookeeper 中镜像根节点的名称，默认 /nodepool，对于多 Nodepool 共享 Zookeeper 服务时比较有意义，可以为每个 Nodepool 配置一个根节点，这样各 Nodepool 间镜像信息彼此独立。

6.5　镜像管理系统

自动制作工具目前有很多，主要有 diskimage-builder、Oz、VMBuilder、VeeWee、Packer 以及 ImageFactory 等，Nodepool 中 Nodepool-builder 使用的制作工具为 diskimage-builder，简称 DIB。

仅供非商业用途或交流学习使用

6.5.1　DIB 介绍

DIB 是 OpenStack 中 TripleO 下的子项目，功能较为完善，支持主流的 Linux 操作系统的镜像制作，包括 fedora、centos、ubuntu 及 debian 等，支持的格式也比较多，如 qcow2、tar、tgz、vhd、docker、aci、raw 及 squashfs 等。

1. DIB 安装

首先安装依赖包 qemu-utils 和 kpartx，如：

```
# apt-get install -y qemu-utils
# apt-get install -y kpartx
```

然后安装 DIB：

```
# pip install diskimage-builder
```

也可以通过源码方式安装 DIB：

```
# git clone https://git.openstack.org/openstack/diskimage-builder
# cd diskimage-builder
# pip install -e .
```

2. DIB 使用

DIB 使用比较简单，只需要通过简单的配置即可使用，如：

```
# disk-image-create -o centos7 -t qcow2 centos7 vm
```

上面的命令运行完毕即可生成 qcow2 格式的 centos7 的镜像，按下面的方式可以生成 raw 格式 Ubuntu14.04 的镜像：

```
# disk-image-create -o ubuntu -t raw ubuntu vm
```

如果需要生成 Ubuntu16.04 镜像，只需要设置 export DIB_RELEASE=xenial 即可。下面是 disk-image-create 的常用参数：

- -a：指定 i386|amd64|armhf 架构类型，默认为 amd64。
- -o imagename：指定输出的镜像名称，默认为镜像的名称。
- -t：取值 qcow2、tar、vhd、docker、aci 或 raw，指定输出的镜像文件格式，默认为 qcow2。
- -x：打开调试信息。
- -u：不压缩镜像，镜像文件较大，但是执行快。
- -c：表示开始工作之前先清理环境。
- --checksum：为创建的镜像文件产生 MD5 和 SHA256 校验和文件。
- --image-size size：设置创建镜像的大小，单位为 GB。
- --image-cache directory：本地缓存镜像的路径，默认为‘~/.cache/image-create’。

- --max-online-resize size：设置支持的最大文件系统块；需要一个真实大小的根分区时候，这个参数很适用。
- --min-tmpfs size：设置建立镜像时 host 上需要的最小 GB 的 tmpfs。
- --mkfs-options：设置 mkfs 命令调用的参数，整个参数为一个字符串。
- --no-tmpfs：不使用 tmpfs 加速镜像创建。
- --offline：不更新缓存资源。
- --qemu-img-options：设置 qemu-img 命令调用的参数。
- --root-label label：指定根文件系统的标签，默认为‘cloudimg-rootfs’。
- --ramdisk-element：指定用于建立镜像的主要元素，默认有‘ramdisk’。
- --install-type：指定默认安装类型，默认为‘source，’也可以设置为‘package’用于包的安装。
- --docker-target：如果输出类型为 docker，则需指定 repo 和 tag，默认为镜像名称。
- -n：跳过默认的‘base’元素选项，RAMDISK 特有的选项。
- -p package[, package, package]：列举在镜像中需要额外安装的包。
- -h | --help：显示帮助信息。
- --version：显示版本信息。

注解 至少需要指定一个 Element，如果使用 VHD 输出格式，还需要安装 vhd-util。

6.5.2 DIB 原理

在介绍 DIB 原理之前，我们先看看 Linux 的文件系统，从工作空间上来说，Linux 文件系统分为 bootfs 和 rootfs，前者位于内核空间，后者位于用户空间，如图 6-10 所示。

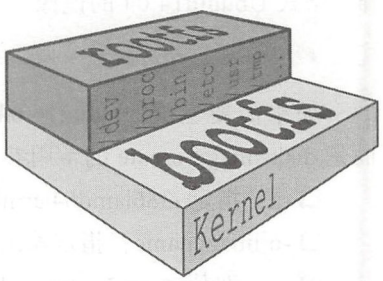


图 6-10 Linux 文件系统

Linux 刚启动时会加载 bootfs 文件系统，然后加载 rootfs，我们熟悉的 /dev、/proc、/bin 等目录就位于 rootfs 中。不同 Linux 发行版的区别主要是 rootfs，比如 Ubuntu 14.04 使用 upstart 管理服务，apt 管理软件包，而 CentOS 7 使用 systemd 和 yum，这些都是用户空间上的区别，内核空间差别不大。而对于一个精简的操作系统，rootfs 可以很小，只需要包括最基本的命令、工具和程序库就可以了，一般只有几十到几百兆。不同版本的 Linux 系统都会发布这样的精简操作系统，并携带 bootfs，我们称其为初始根文件系统，DIB 就是基于这些初始根文件系统增加需要的软件包来制作镜像的。

DIB 的原理如图 6-11 所示。

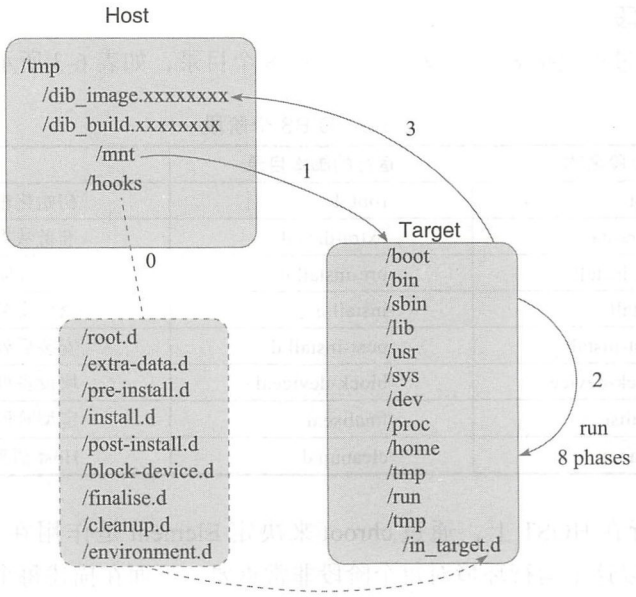


图 6-11　DIB 原理图

镜像制作需要的初始根文件系统，可以从对应操作系统的官网上下载，利用 losetup、qemu-img 和 kpartx 等一系列工具，在 Host（运行 DIB 的系统）中将初始根文件系统和预定义安装配置脚本放到临时目录中，通过 chroot 改变系统的根路径（mnt），然后依次运行预定义的安装配置脚本来定制需要的系统和软件。这些预定义脚本按运行阶段分 8 个目录，都位于 Host 的临时目录 hooks 中，运行时根据最终作用环境，这些目录会映射到目标系统即 Target 的 /tmp/in_target.d 目录下（下文会详细描述这 8 个阶段）。最后将安装了定制软件的目标系统转换成需要的镜像格式，这样一个定制化的镜像就制作成功了。

从以上过程中可以看到，使用 DIB 制作镜像的关键是一些“预定义的安装配置脚本”，这些脚本决定了定制的结果，为了方便管理，DIB 中将这些脚本称为 Element，并按照镜像制作的 8 个阶段规定了 Element 中目录及文件的组成。一个典型的 Element 如图 6-12 所示。

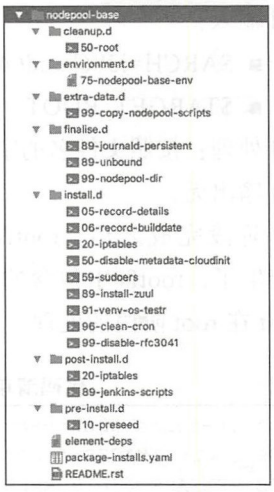


图 6-12　Element 例子——nodepool-base

技巧　基于 DIB 的工作原理，初始文件系统也可以基于 Linux 发行版而不仅仅是精简系统，还可以基于一些特殊场景（比如操作系统内核也是定制，而非来自官方）来制作镜像。

1. DIB 8 个阶段

DIB 镜像制作过程包括 8 个阶段，分别对应 8 个目录，如表 6-3 所示。

表 6-3 DIB8 个阶段

序号	阶段名称	运行的脚本目录	说明
1	root	root.d	初始化根文件系统阶段
2	extra-data	extra-data.d	准备数据阶段
3	pre-install	pre-install.d	预安装阶段
4	install	install.d	软件安装阶段
5	post-install	post-install.d	安装后处理阶段
6	block-device	block-device.d	块设备处理阶段
7	finalise	finalise.d	定型阶段
8	cleanup	cleanup.d	Host 清理阶段

由于 DIB 运行在 HOST 上，通过 chroot 来决定 Element 是作用在目标系统上，还是 HOST 系统上，所以这个运行环境对每个阶段非常重要，下面在描述每个阶段时都会特别加以说明。另外，对于每个阶段如果存在多个 Element 需要执行，则会按照文件名排序，每个阶段的脚本名都会以两个数字开始，用以决定执行顺序。

（1）root 阶段

初始化根文件系统阶段，通常是在已经存在的初始镜像上增加定制的文件内容。

- 运行环境：Host。
- 输入：
  - $ARCH=i386|amd64|armhf|arm64
  - $TARGET_ROOT=/path/to/target/workarea
- 处理：按照文件名的字母顺序依次运行 root.d 目录下的可执行脚本。
- 输出无。

本阶段完成之后，rootfs 需要的 /bin、/sbin、/etc、/lib、/usr 等目录及其中包含的文件就产生了，rootfs 中包含的内容取决于执行的具体脚本。代码清单 6-36 所示的是 ubuntu Element 在 root 阶段的处理。

代码清单 6-36 Element ubuntu 在 root 阶段的处理

```
    # **10-cache-ubuntu-tarball**
    DIB_CLOUD_IMAGES=${DIB_CLOUD_IMAGES:-http://cloud-images.ubuntu.com}
    DIB_RELEASE=${DIB_RELEASE:-trusty}
    BASE_IMAGE_FILE=${BASE_IMAGE_FILE:-$DIB_RELEASE-server-cloudimg-$ARCH-root.tar.gz}
    SHA256SUMS=${SHA256SUMS:-https://${DIB_CLOUD_IMAGES##http?(s)://}/$DIB_RELEASE/current/
SHA256SUMS}
    CACHED_FILE=$DIB_IMAGE_CACHE/$BASE_IMAGE_FILE
    CACHED_FILE_LOCK=$DIB_LOCKFILES/$BASE_IMAGE_FILE.lock
    CACHED_SUMS=$DIB_IMAGE_CACHE/SHA256SUMS.ubuntu.$DIB_RELEASE.$ARCH
```

```
function get_ubuntu_tarball() {
    if [ -n "$DIB_OFFLINE" -a -f "$CACHED_FILE" ] ; then
        echo "Not checking freshness of cached $CACHED_FILE."
    else
        echo "Fetching Base Image"
        $TMP_HOOKS_PATH/bin/cache-url $SHA256SUMS $CACHED_SUMS
        $TMP_HOOKS_PATH/bin/cache-url \
            $DIB_CLOUD_IMAGES/$DIB_RELEASE/current/$BASE_IMAGE_FILE $CACHED_FILE
        pushd $DIB_IMAGE_CACHE
        if ! grep "$BASE_IMAGE_FILE" $CACHED_SUMS | sha256sum --check - ; then
            $TMP_HOOKS_PATH/bin/cache-url -f $SHA256SUMS $CACHED_SUMS
            if ! grep "$BASE_IMAGE_FILE" $CACHED_SUMS | sha256sum --check - ; then
                $TMP_HOOKS_PATH/bin/cache-url -f \
                    $DIB_CLOUD_IMAGES/$DIB_RELEASE/current/$BASE_IMAGE_FILE $CACHED_FILE
                grep "$BASE_IMAGE_FILE" $CACHED_SUMS | sha256sum --check -
            fi
        fi
        popd
    fi
    sudo tar -C $TARGET_ROOT --numeric-owner -xzf $DIB_IMAGE_CACHE/$BASE_IMAGE_FILE
}

(
    echo "Getting $CACHED_FILE_LOCK: $(date)"
    # Wait up to 20 minutes for another process to download
    if ! flock -w 1200 9 ; then
        echo "Did not get $CACHED_FILE_LOCK: $(date)"
        exit 1
    fi
    get_ubuntu_tarball
) 9> $CACHED_FILE_LOCK
```

由代码清单 6-36 可知，ubuntu Element 在 root 阶段主要是根据用户设置的 ubuntu 版本（DIB_RELEASE）从对应的地址（DIB_CLOUD_IMAGES）下载初始根文件系统，缓存初始根文件系统在指定的目录（DIB_IMAGE_CACHE）下。

（2）extra-data 阶段

准备数据阶段。从宿主机中将需要的文件（如 SSH 密钥、http 代理设置等）复制到制作镜像的临时目录 $TMP_HOOKS_PATH 中。

- 运行环境：Host。
- 输入：$TMP_HOOKS_PATH。
- 处理：按照文件名的字母顺序依次运行 extra-data.d 目录下的可执行脚本。
- 输出：无。

本阶段运行完毕后，需要的文件已经从 Host 上复制到 Target 对应的目录中，复制的具体内容和位置取决于执行的脚本。代码清单 6-37 所示的是 Element dpkg 在 extra-data 阶段的处理。

代码清单 6-37 Element dpkg 在 extra-data 阶段的处理

```
# **01-copy-apt-keys**
DIB_ADD_APT_KEYS=${DIB_ADD_APT_KEYS:-""}
if [ -z "${DIB_ADD_APT_KEYS}" ]; then
    echo "DIB_ADD_APT_KEYS is not set - not importing keys"
    exit 0
fi

DIR=${TMP_MOUNT_PATH}/tmp/apt_keys
if [ -e ${DIR} ]; then
    echo "${DIR} already exists!"
    exit 1
fi
sudo mkdir -p ${DIR} # dib-lint: safe_sudo

# Copy to DIR
for KEY in $(find ${DIB_ADD_APT_KEYS} -type f); do
    sudo cp -L ${KEY} ${DIR} # dib-lint: safe_sudo
done
```

由代码清单 6-37 可知，dpkg Element 在 extra-data 阶段定义了一个处理脚本，位于 01-copy-apt-keys 文件中，主要完成的就是复制 apt-keys 到 Target 系统中。

（3）pre-install 阶段

预安装阶段，一般用来准备软件包安装需要用到的仓库，该阶段是在 Target 中运行的第一个阶段。

- 运行环境：Target。
- 输入：无。
- 处理：按照文件名的字母顺序依次运行 pre-install.d 目录下的可执行脚本。
- 输出：无。

本阶段运行完毕后，预安装阶段的准备工作就完成了，需要准备的具体内容取决于具体的执行脚本。需要注意的是，虽然也可以在此阶段做一些包的安装操作，但不建议这么做，包的安装放在下一个阶段进行。代码清单 6-38 所示的是 Element ubuntu 在 pre-install 阶段的处理。

代码清单 6-38 Element ubuntu 在 pre-install 阶段的处理

```
# **00-remove-apt-xapian-index**
apt-get --yes remove apt-xapian-index || :

# **00-remove-grub**
if dpkg-query -W grub-pc ; then
apt-get -y remove grub-pc
fi

if dpkg-query -W grub2-common; then
```

```
        apt-get -y remove grub2-common
    fi

    # **01-set-ubuntu-mirror**
    DIB_DISTRIBUTION_MIRROR=${DIB_DISTRIBUTION_MIRROR:-}

    [ -n "$DIB_DISTRIBUTION_MIRROR" ] || exit 0

    sudo sed -ie "s&http://\(archive\|security\).ubuntu.com/ubuntu&$DIB_DISTRIBUTION_MIRROR&" \
        /etc/apt/sources.list
```

由代码清单 6-38 可知，ubuntu Element 在 pre-install 阶段定义了三个脚本，分别是 00-remove-apt-xapian-index、00-remove-grub 和 01-set-ubuntu-mirror，前面两个脚本是删除不需要的软件，最后一个是设置软件安装的 repo 地址为用户指定的地址。

（4）install 阶段

安装阶段，即安装预定义的软件，镜像绝大多数的包都是在这个阶段处理的，这个阶段的执行通常需要消耗最长的时间。

- 运行环境：Target。
- 输入：无。
- 处理：按照文件名的字母顺序依次运行 install.d 目录下的可执行脚本。
- 输出：无。

本阶段运行完毕后，为镜像定制的绝大多数软件包已经安装完毕，具体需要安装哪些软件包取决于具体定义的脚本和相关定义文件（如 element-deps 等）。Element ubuntu 在 install 阶段的处理如下所示：

```
# **99-autoremove**
apt-get -y autoremove
```

由上可知，ubuntu Element 在安装阶段的最后会清理系统缓存包。

（5）post-install 阶段

安装后处理阶段，包成功安装后，在第一次启动前需要执行的任务都在此阶段处理。例如通过 chkconfig 关闭一些服务，或对安装的服务进行配置等；也可以清理一些缓存的软件包，以减小镜像的大小。

- 运行环境：Target。
- 输入：无。
- 处理：按照文件名的字母顺序依次运行 post-install.d 目录下的可执行脚本。
- 输出：无。

本阶段运行完毕后，定制镜像的文件系统已经基本完成，相关软件的配置也已经准备就绪，具体需要做哪些后处理取决于具体定制的脚本。

技巧 需要注意的是，在此阶段完成后，DIB 会自动运行 fstrim 工具来真正清理“垃圾文件”，释放垃圾文件占用的磁盘空间，有效减小镜像文件的大小，所以如果对最终镜像文件大小有特别的限制，在此阶段需要将不必要的文件删除，以减少镜像大小。

代码清单 6-39 所示的是 Element ironic-agent 在 post-install 阶段的处理。

代码清单 6-39 Element ironic-agent 在 post-install 阶段的处理

```
# **80-ironic-agent**
rm -rf /tmp/ironic-python-agent

case "$DIB_INIT_SYSTEM" in
    upstart)
        if [ -f /etc/init/ufw.conf ]; then
            mv /etc/init/ufw.conf /etc/init/ufw.conf.disabled
        fi
        if [ -f /etc/init/tgt.conf ]; then
            mv /etc/init/tgt.conf /etc/init/tgt.conf.disabled
        fi
        ;;
    systemd)
        if [[ $(systemctl --no-pager list-unit-files iptables) =~ 'enabled' ]]; then
            systemctl disable iptables.service
        fi
        systemctl enable $(svc-map ironic-python-agent).service
        systemctl enable ironic-agent-create-rescue-user.path
        ;;
    sysv)
        update-rc.d iptables disable
        ;;
    *)
        echo "Unsupported init system"
        exit 1
        ;;
esac
```

由代码清单 6-39 可知，ironic-agent 安装完成后，除了删除垃圾文件，还会禁用防火墙。

（6）block-device 阶段

块设备处理阶段，对磁盘进行分区、分区文件系统、挂载系统目录等操作都在此阶段处理。

- 运行环境：Host。
- 输入：
  - $IMAGE_BLOCK_DEVICE={path}
  - $TARGET_ROOT={path}
- 处理：按照文件名的字母顺序依次运行 block-device.d 目录下的可执行脚本。

非卖品，仅供非商业用途或交流学习使用

❑ 输出：$IMAGE_BLOCK_DEVICE={path}。

本阶段运行完毕后，镜像的分区、分区文件系统及挂载系统目录等都会确定。具体需要分几个区、每个区使用什么文件系统、分区挂载的目录以及分区是否自动上电挂载等取决于选择的 Element 中分区文件的定义。代码清单 6-40 所示的是系统默认的 MBR 分区定义。

代码清单 6-40　默认 MBR 分区定义

```
# **block-device-default.yaml**

- local_loop:
    name: image0

-partitioning:
    base: image0
    label: mbr
    partitions:
     - name: root
       flags: [ boot, primary ]
       size: 100%
       mkfs:
            mount:
            mount_point: /
            fstab:
                    options: "defaults"
                    fsck-passno: 1
```

默认的 MBR 分区只定义了 root 一个主分区，主分区也是启动分区，占用整个磁盘空间，并且启动后自动挂载到“/”目录。除了 MRB 分区方式之外，DIB 还支持 GPT 分区方式。

注解　要在镜像制作过程中支持分区操作，除了选择需要的分区 Element（block-device-mbr、block-device-gpt 或 block-device-efi）外，还必须选择 vm 这个 Element，否则 DIB 不会对镜像进行分区操作，而只制作包含根文件系统的镜像。

（7）finalise 阶段

定型阶段，对根文件系统做最后的设置，常常用来设置 SELinux、安装 bootloaders 等。

❑ 运行环境：Target。

❑ 输入：无。

❑ 处理：按照文件名的字母顺序依次运行 finalise.d 目录下的可执行脚本。

❑ 输出：无。

这是修改目标系统的最后一个阶段，所以需要限制在此阶段的软件安装。如果在 finalise 阶段安装了 bootloader，那么此阶段运行完成之后，一个可启动的镜像就制作完毕。

如在制作虚机启动的 VM 时，需要使用 vm 这个 Element，在 finalise 阶段就会处理 SELinux 或安装 bootloader grub2，由于代码较多，有兴趣的读者可以查阅 DIB 中 bootloader

非卖品，仅供非商业用途或交流学习使用

这个 Element 的处理。

（8）cleanup 阶段

清理阶段，一般用来清理 Host 上一些不必要的垃圾文件、包缓存文件或临时代码等。

- 运行环境：Host。
- 输入。
  - $ARCH=i386|amd64|armhf|arm64
  - $TARGET_ROOT=/path/to/target/workarea
- 处理：按照文件名的字母顺序依次运行 cleanup.d 目录下的可执行脚本。
- 输出：无。

此阶段运行完毕后，会根据配置转换镜像格式，即从 raw 格式转换为 qcow2、tar 等用户配置的格式。

2. 其他目录和文件

除了上面提到的 8 个目录外，DIB 还提供了下面的几个目录或文件，有的是 DIB 直接使用的，有的是其他 Element 要求的，下面讲解几个常用的目录和文件。

- environment.d：环境设置目录，放置设置环境变量的脚本，DIB 在上述每个阶段的脚本执行前，都会运行该目录下的脚本。代码清单 6-41 所示的是 Element ubuntu 环境变量的定义。

代码清单 6-41　Element ubuntu 环境变量的定义

```
# **10-ubuntu-distro-name.bash**
export DISTRO_NAME=ubuntu
export DIB_RELEASE=${DIB_RELEASE:-xenial}

# **99-cloud-init-datasources.bash**
export    DIB_CLOUD_INIT_DATASOURCES=${DIB_CLOUD_INIT_DATASOURCES:-"Ec2"}
```

- element-deps：依赖文件，纯文本格式，指明本 Element 依赖的其他 Element 名称，每个依赖一行。如代码清单 6-42 所示是 Element ubuntu 依赖的其他 Elements 定义。

代码清单 6-42　Element ubuntu 使用的其他 Elements

```
cache-url
cloud-init-datasources
dkms
dpkg
ubuntu-common
```

- package-installs.yaml：安装包定义文件，yaml 格式，描述了在 Element 执行过程中需要安装的包。该文件被 package-install Element 使用，所以如果存在该文件，需要在 element 中定义 package-install。定义时除了指明软件包名称外，还需要指定如

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

下信息：

- phase：包操作执行的阶段，默认 install.d。
- uninstall：包是安装还是卸载，false 表示安装，true 表示卸载，默认 false。
- installtype：安装类型，取值 source 或 pkg，默认 pkg 安装方式。
- dib_python_version：Python 版本，2 或 3，如果没有设置则取环境变量 DIB_PYTHON_VERSION 的值。

例如在 ubuntu Element 中，此文件中的定义为 linux-image-generic:，表明在安装基于 Ubuntu 的镜像时，需要在 install 阶段安装 linux-image-generic。

❑ README.rst：一般用来描述该 Element 的功能以及使用说明等。

3. DIB 调试

在制作镜像的过程中可能会遇到一些问题，为了便于排查，DIB 也提供了调测的方法。使用 break 变量可以在特定的情况下，启动一个 shell，让你查看当前镜像的上下文情况。break 点的设置方法一般是：

```
break=[before|after]-hook_name
```

其中 hook_name 是前文介绍的 8 个阶段的名称或 error，如果有多个 break 的点，可以用逗号隔开。下面是设置示例：

```
break=before-block-device,before-finalise
break=before-pre-install
break=after-error
```

其中第一个是在 block-device 和 finalise 阶段前 break，第二个在 pre-install 阶段前 break，第三个是在任何阶段只要出错就会 break。

4. Elements

DIB 提供了非常多的用于制作镜像的 Elements，每个 Element 都是镜像制作中的一个完整功能，如图 6-13 所示是 DIB 提供的部分标准 Elements。

完整的 Elements 的使用请参考官方文档[⊖]。此外，也存在较多第三方项目开发的 Elements，如 tripleo-image-elements、tripleo-puppet-elements 等，这些 Elements 种类齐全、功能丰富，而且都经过充分的测试，是制作镜像的首选。

图 6-13　部分 DIB 标准 Elements

6.5.3　定制镜像

使用 DIB 定制镜像有以下几种方式：

⊖ http://docs.openstack.org/developer/diskimage-builder/elements.html。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

1）使用 DIB 自带的 Elements 定制镜像。

2）使用第三方的 Elements。

3）使用 DIB 工具自带的参数 -p（该参数指定的包会在 install 阶段执行完成后 post-install 阶段执行前安装），但是这种方式有一种局限性，即不能跨系统，因为同一个软件包在不同的系统中名称可能不一样。

4）自开发 Elements，这个也是使用 DIB “最高的境界”，可以完全根据自己的需要定制镜像，灵活度和自由度都非常大，但需要对 DIB 有较深的理解。

由于 DIB 的 Element 非常多，而且几乎没有哪一个 Element 覆盖到 DIB 镜像制作的所有 8 个阶段，所以读者在实现自定义的 Element 时，尽量多参考 DIB 或第三方的 Element 实现的例子，本书就不再举例说明。

6.6 本章小结

本章首先介绍了 Nodepool 的相关背景和安装过程，接着针对 Nodepool 的原理，分析了 Nodepool 的主要流程和实现细节，然后重点阐述了 Nodepool 的配置并给出了相关具体示例，最后简单描述了镜像制作相关的原理。通过对本章内容的学习，期望读者能搭建 Nodepool 环境并熟练使用 Nodepool。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

第 7 章 Chapter 7

日志服务器

本章主要介绍 Jenkins 任务产生的日志文件的相关处理，包括日志服务器的作用以及使用方法等内容。

7.1 日志服务器的作用

Jenkins 每次 Job 运行都会产生一些日志文件，这些文件在默认情况下会保存在 Jenkins Master 服务器上。日积月累之后，Jenkins Master 上会保存大量的日志文件，既占用空间，又可能影响 Jenkins Master 的性能。用户可以在定义 Job 的时候指定日志保存的天数或者构建次数，以删除过期的日志，其配置如图 7-1 所示。

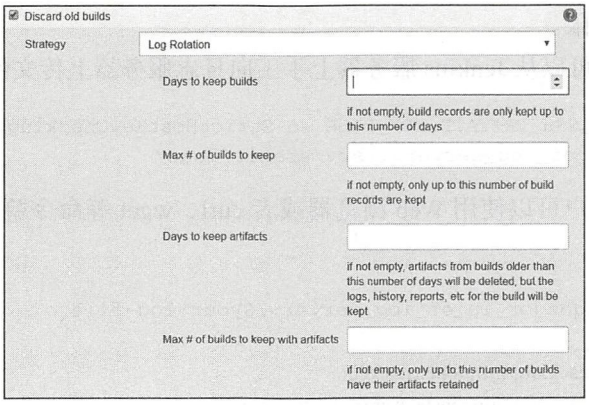


图 7-1 Jenkins 丢弃旧的构建策略

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

Discard old builds 配置项可以帮助 Jenkins 减少日志文件所占的空间，但是，如果用户需要查询这些被删除的构建的日志，就会遇到找不到文件的情况。

Job 运行时，还可能会动态产生一些用户文件，比如用户的代码中生成的数据或文档，这对于用户排查、确认问题可能是有帮助的。这些文件默认存储在运行 Job 的 Slave 上，如果 Slave 是动态的，则删除 Slave 后这些文件也就不存在了；如果 Slave 是静态的，则新的 Job 运行时很可能覆盖或删除上一次运行时产生的文件。

为了持久地保存每次 Job 运行产生的日志和用户文件，OpenStack CI/CD 系统中搭建了专门的日志服务器，来实现这一目的。

7.2 安装和验证

在 OpenStack CI/CD 系统中，日志服务器目前只提供了通过 Puppet 脚本安装的这一种方式。具体的使用方法请参考 OpenStack CI Puppet 模块文档[㊀]。本书中，关于 Puppet 和脚本的使用都在第 10 章中介绍。在参考文档中，首先要通过 system-config[㊁]项目安装 puppet 模块，读者可以查看该项目库的 modules.env 文件获知安装的 puppet 模块的列表，其中，日志服务器相关的安装脚本主要是在 openstack-infra/puppet-openstackci 中。读者可以通过以下方法克隆 openstack-infra/puppet-openstackci 库：

```
git clone https://git.openstack.org/openstack-infra/puppet-openstackci
```

或者通过浏览器访问 openstack-infra/puppet-openstackci 的 git 代码库[㊂]，在线阅读该库中日志服务器的相关代码，包括 log_server_site.pp、logserver.pp 等。

日志服务器安装脚本的主要作用包括：

- 创建 HTTP 服务站点；
- 创建名为 jenkins 的用户，并使其拥有 HTTP 服务主目录的写入权限；
- 信任 Jenkins 所使用 SSH key，使 Jenkins 获得通过 jenkins 用户向日志服务器上 SCP 拷贝文件的权限。

安装之后，用户可以从 Jenkins 服务器上手工向日志服务器上传文件，以验证服务可用：

```
scp -i $JENKINS_SSH_PRIVATE_KEY_FILE -o StrictHostKeyChecking=no $your-log-file \
jenkins@<fqdn_or_ip_of_log_server>:/srv/static/logs/
```

上传成功后，用户可以使用 Web 浏览器或者 curl、wget 等命令验证自己上传的文件是否存在。例如：

```
curl    http://<fqdn_or_ip_of_log_server>/$your-log-file
```

㊀ https://docs.openstack.org/infra/openstackci/。

㊁ https://git.openstack.org/cgit/openstack-infra/system-config。

㊂ https://git.openstack.org/cgit/openstack-infra/puppet-openstackci。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

7.3　使用方法

7.3.1　在 Jenkins 中使用日志服务器

日志服务器搭建之后，Jenkins Job 就可以通过 SCP 的方式将日志和文件放置到日志服务器上保存了。首先，Jenkins 需要安装和配置 SCP 相关的插件，具体方式请参考第 4 章。

然后，用户在定义自己的 Job 时，在 Job 配置中的 Post-build Actions 部分，增加 Publish artifacts to SCP Repository 的配置。下面给出两个例子进行说明。

在图 7-2 中，文件来源配置为空，但选中了“Copy Console Log”选项，即该 Job 运行后会把控制台的输出日志通过 SCP 拷贝到日志服务器上。

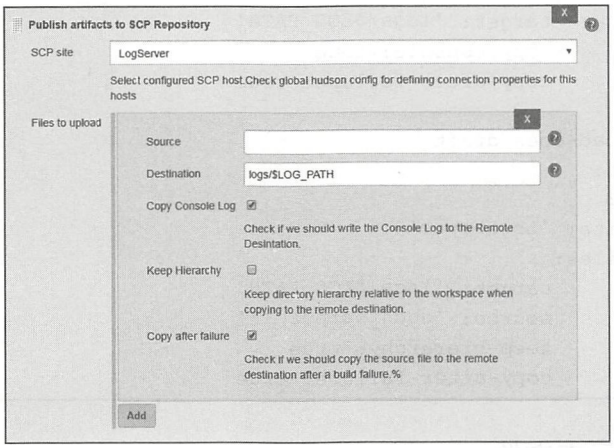


图 7-2　Jenkins Job 拷贝控制台日志

在图 7-3 中，文件来源为 docs_output/**，即该 Job 运行后会把工作目录下的 docs_output/** 子目录的所有文件通过 SCP 拷贝到日志服务器上。

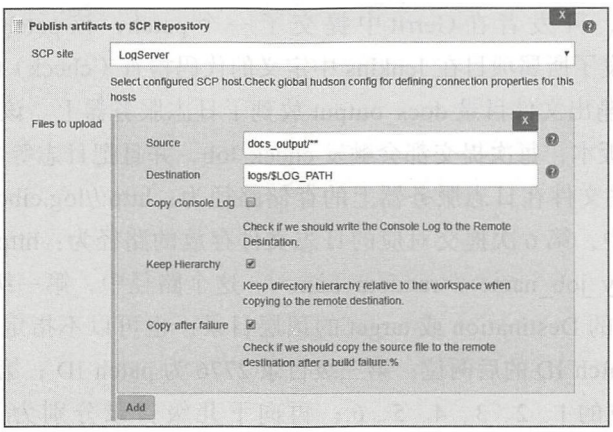


图 7-3　Jenkins Job 拷贝文档文件

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

用户可以点击图 7-3 中的 Add 按钮以增加配置，在一个 publisher 中拷贝多个来源的文件。

如果用户使用 JJB 定义 Job，而不是使用 Jenkins 的 Web 界面，则可以参考代码清单 7-1 定义自己的 publisher。

代码清单 7-1　使用 Jenkins publisher 拷贝日志

```
- publisher:
    name: console-log
    publishers:
        - scp:
            site: 'LogServer'
            files:
                - target: 'logs/$LOG_PATH'
                  copy-console: true
                  copy-after-failure: true
- publisher:
    name: upload-docs-draft
    publishers:
        - scp:
            site: 'LogServer'
            files:
                - target: 'logs/$LOG_PATH'
                  source: 'docs_output/**'
                  keep-hierarchy: true
                  copy-after-failure: true
```

7.3.2　如何获取日志文件

经过了上一节中介绍的配置过程，Jenkins Job 在运行之后，会自动把日志等文件存储到日志服务器上。但是，使用者还需要能够获取到这些文件，而且能够把这些文件与具体的 Job 或者 Git/Gerrit 上的 patch 对应起来，以方便使用。

如图 7-4 所示，开发者在 Gerrit 中提交了一个 patch，该 patch 在 Gerrit 中的 ID 为 2776。该次提交触发了所属项目在 Jenkins 中定义的代码检查（check）的 Job，check Job 运行完成后把日志和输出文件目录 docs_output 放到了日志服务器上。该 patch 经过了多次修改，共提交了 6 个版本，每次提交都会触发 check Job，并且把日志等文件放到日志服务器上。该 patch 的相关文件在日志服务器上的存储路径为：http://log.cibook.oz/logs/76/2776/，如图 7-4 所示。其中，第 6 次提交对应的日志文件存放的路径为：http://log.cibook.oz/logs/76/2776/6/check/<my_job_name>/<random_num>/。这个路径中，第一级目录 logs 为上一节中 Publisher 中指定的 Destination 或 target 的顶层目录，也可以不指定或者指定多个层次；第二级目录 76 为 patch ID 的后两位；第三级目录 2776 为 patch ID；第四级目录为 patch 的版本号，即上图中的 1、2、3、4、5、6；再向下几级目录分别为 Job 类型（本例中为 check）、Job 名称、随机数。在该路径下，用户可以找到一个 console.html 文件和一个 docs_

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

output 目录，其中 console.html 文件的内容就是 Jenkins Job 运行时的控制台输出日志，docs_output 目录为 Job 的 Publisher 中指定的 Source 目录，被原样拷贝到了日志服务器上。在 check Job 运行结束后，门控系统 Zuul 会把上述路径填写到该 patch 的 comment 中，用户点击该 HTTP 链接就可以看到相关 Job 的运行结果及输出的用户文件目录。

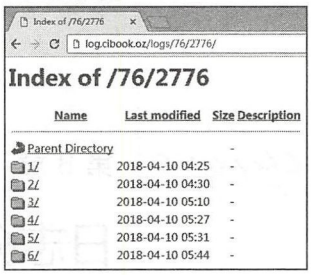


图 7-4　日志存储路径

对于非 Gerrit patch 触发的 Job，比如周期类型的 Job，在日志服务器上存在名为 periodic、periodic-weekly、periodic-monthly 等目录，用户可以到这些目录中以 Job 名称命名的子目录下查找相关的日志。

7.3.3　日志文件定期归档和清理

日志文件存放之后，日志服务器会定期执行 log_archive_maintenance.sh 脚本对文件进行扫描检查，对存放时间较长的文件进行归档和清理。用户可以登录到日志服务器上，以 root 用户执行 crontab -l 来查看相关的定时任务。默认情况下的操作策略为：

- 每星期定时运行归档操作；
- 对扩展名为 .txt、.html 的文件和以 tmp 开头的文件或目录调用 gzip 进行压缩；
- 删除存放超过 30 天的所有文件；
- 删除空目录。

如果用户需要访问已经被压缩但尚未被删除的日志，则可以在目标 URL 后面加上 .gz 之后再尝试。例如，上星期保存了一个日志文件，其 URL 之前是：http://mylogserver/mylogfile.txt，在被压缩之后，可以通过 http://mylogserver/mylogfile.txt.gz 进行访问。被压缩之后又存放超过 30 天的文件也会被删除。被删除后，文件将彻底无法访问。

由于日志服务器是使用 OpenStack 社区提供的 Puppet 脚本安装的，如果需要在安装时就修改该定时归档和清理任务，或者禁止该任务，则可以在安装日志服务器之前，对相关脚本进行修改，包括 logserver.pp、log_archive_maintenance.sh 等，这些脚本都在 openstack-infra/puppet-openstackci[⊖]库中。如果在日志服务器安装之后需要修改该任务，则可以登录到日志服务器上，以 root 用户执行 crontab -e 编辑该定时任务，或者修改归档脚本的内容，该脚本存放的路径为：/usr/local/sbin/log_archive_maintenance.sh。

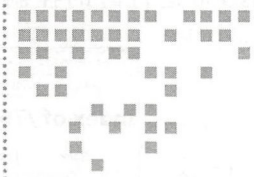
7.4　本章小结

本章对日志服务器的使用方法进行了介绍。解决了日志的存储问题后，对数据进行分析的方法请参考第 8 章。

⊖ https://git.openstack.org/cgit/openstack-infra/puppet-openstackci。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用



Chapter 8 第 8 章

日志分析系统

第 7 章提到的日志服务器解决了日志的集中管理问题。接下来要面对的状况是，面对如此大量的日志，如何高效地查询、排序和统计这些日志？你会发现我们急需一个日志分析系统。

OpenStack CI/CD 的日志分析系统是在 2013 年开发的，彼时 ELK Stack（2016 年更名为 Elastic Stack）以其开源属性和相对完善的功能，成为开源界实时日志处理领域的第一选择。这套日志分析系统是一个很好的 ELK Stack 应用实例，很多地方值得借鉴与学习。

注解
- 2016 年 ELK Stack 在其 5.0 版本加入 Beats（数据收集平台）后更名为 Elastic Stack。
- 下文中提到的日志分析系统如不进行特别说明均指 OpenStack CI/CD 的日志分析系统。

8.1 ELK Stack 概况

最初为大家所知的是 ELK。ELK 并不是一个软件，而是三款开源软件的首字母组合：Elasticsearch、Logstash 和 Kibana，这三款软件各司其职，完美配合，形成了一套完整的日志分析系统。

- Elasticsearch 是基于 Apache Lucene 构建的开源的分布式搜索和分析引擎，具有高可伸缩、高可靠和易管理等特点。

仅供非商业用途或交流学习使用

- Logstash 是一个具有实时工作流功能的开源数据收集引擎。它支持动态地从各种数据源搜集数据，并对数据进行过滤、分析和统一格式等操作，然后存储到用户指定的后端存储区。
- Kibana 是一个专门为 Elasticsearch 服务的开源分析及可视化平台，用来搜索、展示存储在 Elasticsearch 中的数据。

2013 年，Elasticsearch 所属的 Elastic.co 公司先后收购了 Kibana 和 Logstash，ELK stack（后面简称为 ELK）正式成为官方用语。

8.2　日志分析系统架构

日志分析系统是基于 ELK 的，所以我们先来了解下典型的 ELK 架构。图 8-1 是典型的 ELK 架构。

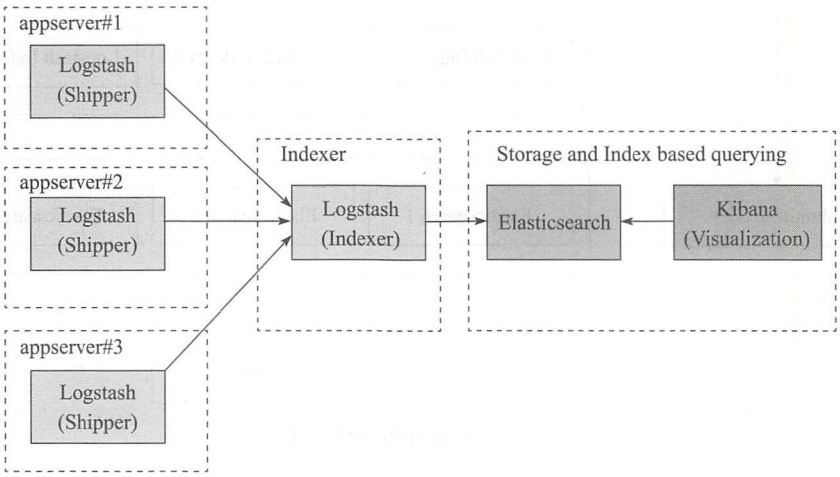


图 8-1　典型的 ELK 架构

整个架构可以划分为四个部分。

1）数据监控与收集：数据源所在的系统中会运行 Logstash（常被称为 Logstash Shipper），用来对数据进行监控与收集，最后运送到 Logstash 服务器（常被称为 Logstash Indexer）。

2）数据预处理：进入到 Logstash Indexer 的数据，会按照要求进行过滤和修改转化为归一化的数据，并送入到 Elasticsearch 集群进行存储。

3）数据索引与搜索：数据在 Elasticsearch 中进行索引，并且 Elasticsearch 还提供强大的搜索功能。

4）数据可视化：通过 Kibana 的 WebUI 界面来查询和展示存储在 Elasticsearch 中的数据。

图 8-2 是日志分析系统架构。

仅供非商业用途或交流学习使用

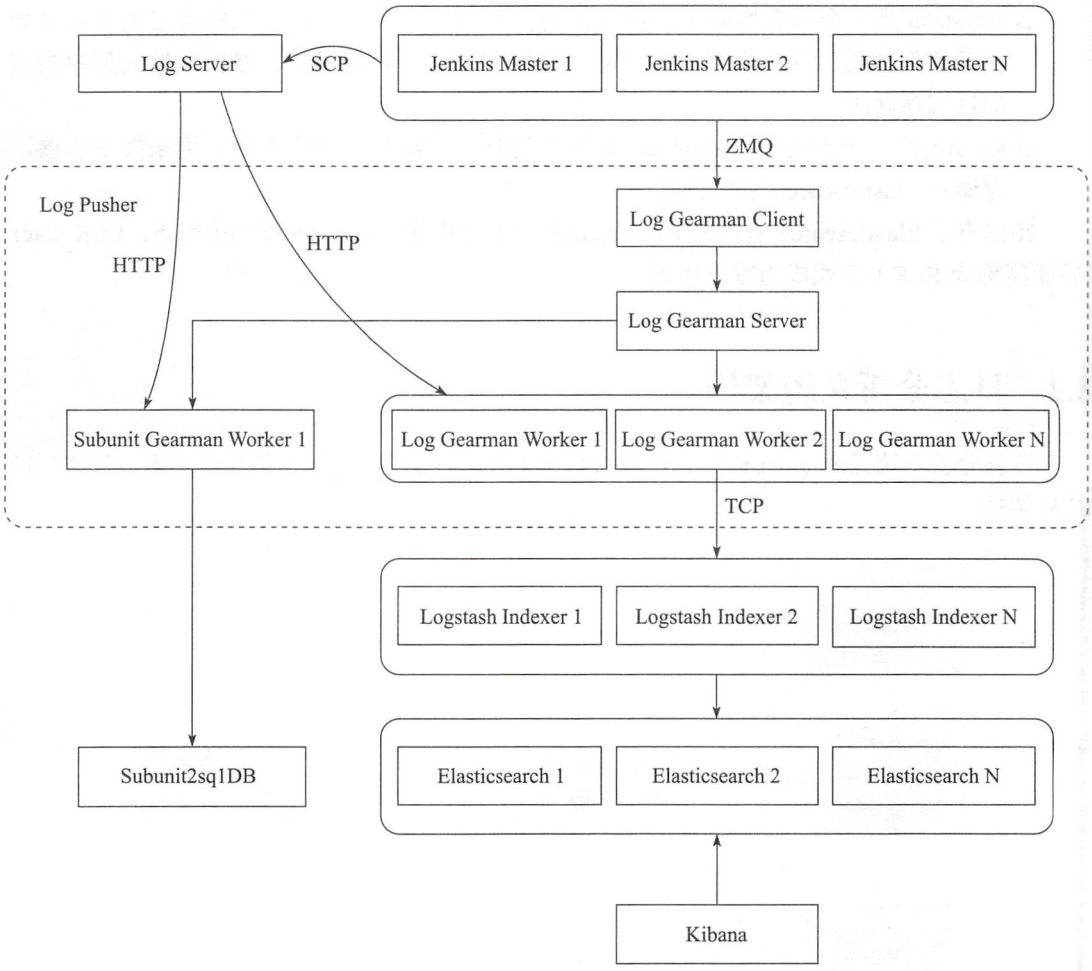


图 8-2　日志分析系统架构

日志分析系统的架构与典型的 ELK 架构总体上是一致的，唯一的差别在于数据的监控与收集这部分所采用的具体技术不同。同样，日志分析系统也能划分为四个部分：

- Log Pusher（数据监控与收集）
- Logstash Indexer（数据预处理）
- Elasticsearch 集群（数据索引与搜索）
- Kibana（数据可视化）

下面将依次对这四个部分进行详细介绍。

图 8-2 中的 Subunit Gearman Worker 专门负责将门限测试结果和周期性 tempest 测试结果存储到 SQL 数据库。这部分不涉及 ELK，故本章不对这部分进行陈述。

8.3　Log Pusher

Log Pusher 是 OpenStack Infra 团队开发的。它采用了 Gearman 架构，随着日志不断增加，也能很容易地进行水平扩展。

8.3.1　处理流程

Log Pusher 的处理流程如下：

1）Log Gearman Client 负责监听 Jenkins ZeroMQ Publisher 发出的 Job 构建完成事件；

2）当监听到构建完成事件后，Log Gearman Client 会向 Gearman Server 提交一个推送日志文件到 Logstash Indexer 的 Gearman 任务；

3）Gearman Server 将此任务分发给一个 Log Gearman Worker；

4）Log Gearman Worker 通过 HTTP 从日志服务器获取日志，然后将日志内容发布到 Logstash Indexer 的 9999 端口；

5）Logstash Indexer 监听 9999 端口获得日志信息。

8.3.2　配置

Log Pusher 的具体实现是通过两个 Python 脚本来完成的。这两个脚本是开源的，可从参考地址[㊀㊁]获得。

对于使用者来说，可以先不了解这两个 Python 脚本的代码细节，但需要清楚脚本使用的配置文件，根据实际需要进行合理配置。

1. jenkins-log-client.yaml

该文件是 log-gearman-client.py 的配置文件，内容如代码清单 8-1 所示。

代码清单 8-1　jenkins-log-client.yaml

```
source-url: http://log.cibook.oz

zmq-publishers:
    - tcp://jenkins.cibook.oz:8888

subunit-files: []

source-files:
    - name: console.html
      retry-get: True
      tags:
        - console
    - name: logs/syslog.txt
```

㊀ https://git.openstack.org/cgit/openstack-infra/puppet-log_processor/tree/files/log-gearman-client.py。

㊁ https://git.openstack.org/cgit/openstack-infra/puppet-log_processor/tree/files/log-gearman-worker.py。

```
    tags:
      - syslog
  - name: logs/libvirtd.txt
    tags:
      - libvirt
```

- source-url：日志文件所在的服务器的访问地址。
- zmq-publishers：监听的 Jenkins ZeroMQ Publisher 地址列表。
- source-files：被监控文件列表，每个文件需给定标识列表属性 tags。

2. jenkins-log-worker.yaml

该文件是 log-gearman-worker.py 的配置文件。内容如代码清单 8-2 所示。

代码清单 8-2 jenkins-log-worker.yaml

```
gearman-host: logstash.cibook.oz
gearman-port: 4730
output-host: localhost
output-port: 9999
output-mode: tcp
```

- gearman-host 和 gearman-port 分别是 Gearman server 地址与端口信息。
- 日志会推送到 <output-mode>://<output-host>:<out-port>，这三个配置参数用于与 Logstash 对接。

8.4 Logstash Indexer

日志并不适合直接存储到 Elasticsearch 中，在进行存储前我们更希望对日志文件进行一些必要的预处理，如删除不想要的信息、修改信息格式等，这些 Logstash Indexer 都能帮我们完成。

8.4.1 hello world

我们通过一个简单的“hello world”例子来直观地看看 Logstash 能干些什么。Logstash 可以通过命令行来运行：

```
bin/logstash -e 'input { stdin { } } output { stdout { codec=>rubydebug } }'
```

“-e”是 Logstash 的参数之一，表示从命令行配置 Logstash。后面跟着的参数就是执行 Logstash 使用的配置，这个配置的意思是：通过标准输入 stdin 读取数据，按照设定编码格式 rubydebug 将结果进行标准输出 stdout。

上面的命令行执行确认后，会在屏幕上看到“Pipeline main started”。在命令提示符处输入“hello world”：

```
hello world
```

确认后，屏幕上会打印出返回结果，如代码清单 8-3 所示：

代码清单 8-3　Logstash 返回结果

```
{
          "host" => "localhost",
      "@version" => "1",
    "@timestamp" => 2018-03-28T08:25:07.769Z,
       "message" => "hello world"
}
```

代码清单 8-3 就是最常见的 Logstash 输出。

- message：输入的数据（事件）。
- @timestamp：事件发生的时间戳。
- host：产生此事件的机器。

简单来说，Logstash 就像一条管道。一条非格式化的数据经过这条管道后，被转化成了一条格式化的事件（转化后的数据在 Logstash 中也被称作事件），这样的事件更便于存储和检索。

8.4.2　Logstash 管道

Logstash 管道有两个必经的处理环节，即输入（input）和输出（output）。此外，还有一个可选的环节——过滤（filter），用于进行数据的过滤和修改，Logstash 的强大之处也体现在其过滤上。Logstash 还在 1.3.0 版本中引入了 Codec 概念。引入后，可以在输入和输出时对数据进行编码或解码处理，如图 8-3 所示。

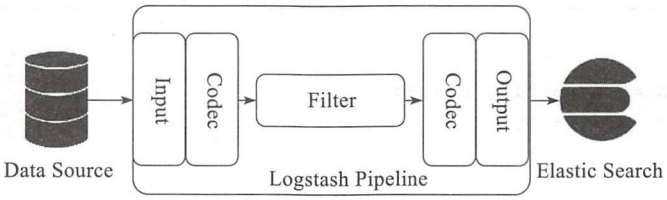


图 8-3　Logstash 管道

数据的具体处理是由插件来完成的。Logstash 提供了丰富的插件，插件按照功能可分为输入插件、过滤器插件、输出插件和编解码器插件[⊖]。用户也可以自己开发插件，在此不做赘述。

8.4.3　管道配置

除了如 hello world 示例，通过 -e 来显示的配置 Logstash 管道，也可以将配置固化在扩

⊖ Logstash Versioned Plugin Reference, https://www.elastic.co/guide/en/logstash-versioned-plugins/current/。

展名为 .conf 的管道配置文件中，通过参数 -f 来告诉 Logstash。

在 Debian 系统中，管道配置文件的默认路径是 /etc/logstash/conf.d。Logstash 会尝试加载这个目录下所有扩展名为 .conf 的文件，拼接出一个完整的配置文件。

1. 管道配置文件结构

Logstash 用 {} 来定义区段。管道配置文件为要使用到的每种类型的插件提供一个独立的区段，如代码清单 8-4 所示。

代码清单 8-4　管道配置文件结构

```
input {
   ...
}

filter {
   ...
}

output {
   ...
}
```

每个区段内可以包含一个或多个插件区段。需要指出的是，如果在过滤器区段内指定了多个过滤器插件，则按照顺序依次应用。插件区段是对具体插件的配置，包含插件名称和该插件的一组设置。代码清单 8-5 为在输入区段配置了两个输入插件（file 插件）。

代码清单 8-5　输入插件区段配置

```
input {
    file {
        path => "/var/log/messages"
        type => "syslog"
    }

    file {
        path => "/var/log/apache/access.log"
            type => "apache"
        }
    }
```

file 插件区段内配置了两个参数：路径 path 和类型 type。其含义如下：

- 监听 /var/log/messages 文件，标注其类型为 syslog。
- 监听 /var/log/apache/access.log，标注其类型为 apache。

插件的配置参数因插件类型而异，可查看对应插件的文档。

2. 配置语法

Logstash 设计了一套自己的 DSL（Domain Specific Language）。除了上面提到的区段，其他语法还包括：

- 数据类型
- 字段引用
- 条件判断

语法细节可以参看官方文档[⊖]，在此不做赘述。

8.4.4　管道配置实例

Logstash 在使用上的一个重点是如何配置管道配置文件，下面介绍一下在日志分析系统使用的管道配置文件。

1. 输入配置文件（input.conf）

输入配置文件内容如代码清单 8-6 所示。

代码清单 8-6　输入配置文件

```
input {
    tcp {
        host => "localhost"
        port => 9999
        codec => json_lines {}
        type => "jenkins"
    }
}
```

- host 和 port 与代码清单 8-2 jenkins-log-worker.yaml 中的 output-host 和 output-port 必须保持一致，否则无法接收到日志信息。
- 对收到的日志使用编解码插件 json_lines 进行解码（codec=>json_lines{}），并标记日志类型为 jenkins（type=>"jenkins"）。

2. 输出配置文件（output.conf）

输出配置文件内容如代码清单 8-7 所示：

代码清单 8-7　输出配置文件

```
output {
    elasticsearch {
        hosts => ['es.cibook.oz:9200']
        manage_template => false
        false timeout => 300
    }
}
```

⊖ Configuring Logstash, https://www.elastic.co/guide/en/logstash/current/configuration.html。

这里后端存储选择的是 elasticsearch，因此使用的输出插件是 elasticsearch。

- hosts：对接的是 elasticsearch 节点地址列表。
- manage_template：此处配置为 false，表示关闭 Logstash 自动创建索引模板功能。
- timeout：发送到 Elasticsearch 的请求响应超时时间。超时后，请求会重发。

3. 过滤配置文件（openstack-filters.conf）

过滤配置文件描述了对日志内容处理的步骤，可参考相关地址[⊖]。根据日志的 tags 值（在 jenkins-log-client.yaml 中定义）不同，进行不同的过滤处理。

以对 console 日志的处理这段代码为例进行说明，如代码清单 8-8 所示。

代码清单 8-8　console 日志处理

```
    if "console" in [tags] or "console.html" in [tags] {
        if [message] == "<pre>" or [message] == "<pre>" {
            drop {}
        }
        multiline {
            negate => true
            pattern => "^%{TIMESTAMP_ISO8601} \|"
            what => "previous"
            stream_identity => "%{host}.%{filename}"
        }
        grok {
            # Do multiline matching as the above mutliline filter may add newlines
            # to the log messages.
            match => { "message" => "(?m)^%{TIMESTAMP_ISO8601:logdate} \| %{GREEDYDATA:
logmessage}" }
            add_field => { "received_at" => "%{@timestamp}" }
        }
    }
```

1）tags 字段包含 console 或 console.html 的事件将进入这个处理分支。

2）首先使用过滤器插件 drop 丢弃 message 字段是 <pre> 或 </pre> 的事件。

3）接下来是编解码插件 multiline。有时调试类日志会为一个事件打印多行内容，此插件能将多行消息合并到一个 Logstash 事件中。事件的 message 字段与 pattern 进行匹配，如果匹配不成功，表明是多行消息中的一行，与上一行进行合并，如果匹配成功，则不进行合并操作。

4）最后使用 grok 插件。这个插件是 Logstash 最重要的插件。事件的 message 字段与 match 中给定的正则表达式进行匹配，匹配到 TIMESTAMP_ISO8601 的部分赋值给 logdate 字段，匹配到 GREEDYDATA 的部分赋值给 logmessage 字段，这两个字段会放入最后的输出结果中。此外，通过配置参数 add_field 往输出结果中加入 received_at 字段，其值为事件发生的时间戳。

⊖ https://github.com/openstack-infra/logstash-filters/blob/master/filters/openstack-filters.conf。

注解　TIMESTAMP_ISO8601 和 GREEDYDATA 为官方提供预定义的 grok 表达式，具体定义可以在 $logstash/vendor/bundle/jruby/2.3.0/gems/logstash-patterns-core-4.1.2/patterns/ 目录下查找。

8.5　Elasticsearch

预处理好的日志事件被送入 Elasticsearch 进行索引。Elasticsearch 使用 Java 开发并使用 Lucene 作为其核心来实现所有索引和搜索的功能。而且它还提供了 RESTfull API 接口来隐藏 Lucene 的复杂性。

Elasticsearch 上手非常简单。它提供了很好的默认设置，对初学者而言，它开箱即用，无须立马面对复杂的理论。本节会涵盖 Elasticsearch 的核心概念、如何进行索引和简单的搜索及使用中涉及的重要配置等内容。

注解　使用 RESTful API 和 Elasticsearch 进行通信，默认端口 9200，也可在配置文件 elastic-search.yml 中进行修改。你可以用浏览器访问 Elasticsearch，在下文中你将看到也能使用 curl 命令来和 Elasticsearch 交互。

8.5.1　面向文档的数据库

Elasticsearch 是面向文档的数据库。它的存储对象是文档。简单来说，文档是序列化的 JSON（JavaScript Object Notation）数据，代码清单 8-9 是一个 JSON 文档，它代表了一个用户对象。

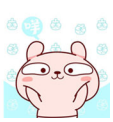
代码清单 8-9　JSON 文档

```
{
    "email": "john@smith.com",
    "first_name": "John",
    "last_name": "Smith",
    "age": 25,
    "interests": [ "dolphins", "whales" ],
    "about": "I love to go rock climbing."
    "join_date": "2014/05/01"
}
```

文档里的基本单元是字段（field），如 "last_name":"Smith"。

在我们把这个 JSON 文档索引到 Elasticsearch 之前，还需要了解两个概念：索引（Index）和类型（Type）。

在 Elasticsearch 中，文档属于一种类型，类型存在于索引中。它们之间的关系如图 8-4 所示。



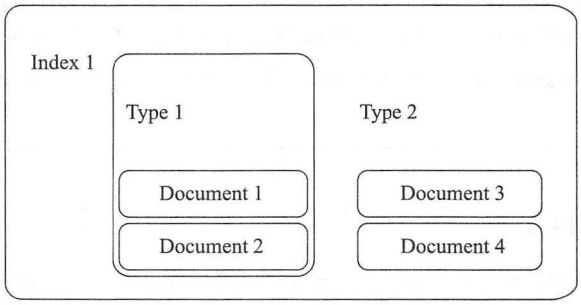


图 8-4　索引、类型和文档的关系

所以在索引文档之前，我们需要确定将文档放入哪个索引的哪个类型下。

8.5.2　索引、检索和搜索

1. 索引文档

现在我们来索引代码清单 8-9 的这个 JSON 文档。实际操作如代码清单 8-10 所示。

代码清单 8-10　索引文档

```
curl -XPUT 'localhost:9200/users/info/1' -d '
{
    "email": "john@smith.com",
    "first_name": "John",
    "last_name": "Smith",
    "age": 25,
    "interests": [ "dolphins", "whales" ],
    "about": "I love to go rock climbing.",
    "join_date": "2014/05/01"
}'
```

URL 中的 /users/info/1 表示文档归属在索引 users 的类型 info 下，且唯一编号为 1。其中，文档的唯一编号可以自己指定，也可让系统自动分配。

现在我们来回看代码清单 8-7，也许你会发现，在 Logstash 输出插件 elasticsearch 中没有出现索引和类型的信息，那么日志被存到哪个索引和哪个类型了呢？ Elasticsearch 输出插件对应索引和类型的配置参数是 index 和 document_type。

- index：使用了默认值 logstash-%{+YYYY.MM.dd}，也就是说一天中的所有日志都索引到同一个索引下，索引名与日期绑定。
- document_type：如果没有配置，对于 Elasticsearch 5.0 及以下版本，优先使用事件的 type 字段，如果没有配置 type 字段，则使用 doc；对于 Elasticsearch 6.0 及以上版本，则直接使用 doc。

2. 检索文档

在 Elasticsearch 中检索出数据也非常简单，实际操作如代码清单 8-11 所示。

非卖品，仅供非商业用途或交流学习使用

代码清单 8-11　检索文档

```
$ curl -XGET 'localhost:9200/users/info/1'
{
    "_index" : "users",
    "_type" : "info",
    "_id" : "1",
    "_version" : 1,
    "found" : true,
    "_source" : {
    "email" : "john@smith.com",
    "first_name" : "John",
    "last_name" : "Smith",
    "age" : 25,
    "interests" : [
        "dolphins",
        "whales"
    ],
        "about" : "I love to go rock climbing.",
        "join_date" : "2014/05/01"
    }
}
```

检索时使用的 URL 与刚刚创建时的一样，只是方法从 PUT 变成了 GET。从返回的信息，我们可以看到除了放在 _source 中原有的 JSON 文档，还包含了一些文档的元数据（metadata）。

- _index：文档的索引。
- _type：文档的类型。
- _id：文档的唯一编号。
- _version：文档版本信息，记录文档变更次数。

3. 搜索文档

下面我们来看看如何使用 Elasticsearch 进行搜索。

第一个尝试的是最简单的搜索——搜索所有用户。搜索使用的是 _search 接口，实际操作如代码清单 8-12 所示。

代码清单 8-12　搜索所有用户

```
$ curl -XGET 'localhost:9200/users/info/_search'
{
        "took" : 3,
        "timed_out" : false,
        "_shards" : {
        "total" : 5,
        "successful" : 5,
        "skipped" : 0,
        "failed" : 0
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
    },
    "hits" : {
        "total" : 2,
        "max_score" : 1.0,
        "hits" : [
            {
                "_index" : "users",
                "_type" : "info",
                "_id" : "2",
                "_score" : 1.0,
                "_source" : {
                    "email" : "jane@smith.com",
                    "first_name" : "Jane",
                    "last_name" : "Smith",
                    "age" : 32,
                    "about" : "I like to collect rock albums",
                    "interests" : [
                        "music"
                    ],
                    "join_date" : "2017/06/01"
                }
            },
            {
                "_index" : "users",
                "_type" : "info",
                "_id" : "1",
                "_score" : 1.0,
                "_source" : {
                    "email" : "john@smith.com",
                    "first_name" : "John",
                    "last_name" : "Smith",
                    "age" : 25,
                    "interests" : [
                        "dolphins",
                        "whales"
                    ],
                    "about" : "I love to go rock climbing.",
                    "join_date" : "2014/05/01"
                }
            }
        ]
    }
}
```

返回结果中各项含义如下。

- took：执行搜索所耗费的时间，单位为毫秒。
- timeout：查询是否超时。通常，搜索请求不会超时。如果相比完整的结果，你更需要的是快速的响应时间，你可以指定在搜索请求的 URL 里指定 timeout 值，例如：

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
/_search?timeout=10ms
```

Elasticsearch 会尽可能地返回指定时间内它所查到的内容。

- _shards：参与查询分片的总数。分片的内容可参见 8.5.4 节。
- hits：返回结果中最重要的内容就是 hits 它包含了匹配文档的总数（hits.total:2），还包含了文档本身，被放在数组 hits.hits 中。每个匹配的搜索结果还有一个 _score 字段。这个是相关性评分，这个数值表示当前文档与查询的匹配程度。通常来说，搜索结果会先返回最匹配的文档，也就是 hits.hits 数组会按照 _score 的高低进行排序。hits.max_score 会显示所有匹配文档中的 _score 的最大值。

接下来进行一个具体点的搜索——搜索所有姓氏为 Smith 的用户，如代码清单 8-13 所示。

代码清单 8-13　使用查询短语进行搜索

```
$ curl -XGET 'localhost:9200/users/info/_search?q=last_name:Smith'
{
    ... 部分内容省略 ...
    "hits" : {
        "total" : 2,
        "max_score" : 0.2876821,
        "hits" : [
            {
                "_index": "users",
                "_type": "info",
                "_id": "2",
                "_score": 0.2876821,
                "_source": {
                    "email": "jane@smith.com",
                    "first_name": "Jane",
                    "last_name": "Smith",
                    "age": 32,
                    "about": "I like to collect rock albums",
                    "interests": [ "music" ],
                    "join_date": "2017/06/01"
                }
            },
            {
                "_index" : "users",
                "_type" : "info",
                "_id" : "1",
                "_score" : 0.2876821,
                "_source" : {
                    "email" : "john@smith.com",
                    "first_name" : "John",
                    "last_name" : "Smith",
                    "age" : 25,
                    "interests" : [
                    "dolphins",
                    "whales"
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
                ],
                "about" : "I love to go rock climbing.",
                "join_date" : "2014/05/01"
                }
            }
        ]
    }
}
```

这里用到的是精简版的搜索，使用的是查询语句（query string），即 URL 中的 q=last_name:Smith。

Elasticsearch 还提供了一个使用 JSON 表示请求体（request body），这种富搜索语言叫作结构化查询语句（DSL），它支持构建更加复杂和健壮的查询。下面我们使用 DSL 进行一个稍微复杂点的搜索，依旧是搜索姓氏为 Smith 的用户，但这次只需要搜索年龄大于 30 的，如代码清单 8-14 所示。

代码清单 8-14　使用 DSL 搜索

```
$ curl -XGET 'localhost:9200/users/info/_search?' -d '
{
    "query" : {
        "bool": {
            "must": {
                "match" : {
                    "last_name": "Smith"
                }
            },
            "filter": {
                "range" : {
                    "age" : { "gt": 30 }
                }
            }
        }
    }
}'
{
    ... 部分内容省略 ...
    "hits" : {
        "total" : 1,
        "max_score" : 0.2876821,
        "hits" : [
            {
                "_index" : "users",
                "_type" : "info",
                "_id" : "2",
                "_score" : 0.2876821,
                "_source" : {
                    "email" : "jane@smith.com",
                    "first_name" : "Jane",
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
                    "last_name" : "Smith",
                    "age" : 32,
                    "about" : "I like to collect rock albums",
                    "interests" : [
                        "music"
                    ],
                    "join_date" : "2017/06/01"
                }
            }
        ]
    }
}
```

现在返回结果中只会显示 32 岁的名为 Jane Smith 的用户。

最后一个示例，我们介绍下 Elasticsearch 的全文搜索功能，搜索所有喜欢 rock climbing 的用户，如代码清单 8-15 所示。

代码清单 8-15　全文搜索

```
$ curl -XGET 'localhost:9200/users/info/_search' -d '
{
    "query": {
        "match": {
            "about": "rock climbing"
        }
    }
}'
{
    ... 部分内容省略 ...
    "hits" : {
        "total" : 2,
        "max_score" : 0.5753642,
        "hits" : [
            {
                "_index" : "users",
                "_type" : "info",
                "_id" : "1",
                "_score" : 0.5753642,
                "_source" : {
                    "email" : "john@smith.com",
                    "first_name" : "John",
                    "last_name" : "Smith",
                    "age" : 25,
                    "interests" : [
                    "dolphins",
                        "whales"
                    ],
                    "about" : "I love to go rock climbing.",
                    "join_date" : "2014/05/01"
                }
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
            },
            {
                "_index" : "users",
                "_type" : "info",
                "_id" : "2",
                "_score" : 0.2876821,
                "_source" : {
                    "email" : "jane@smith.com",
                    "first_name" : "Jane",
                    "last_name" : "Smith",
                    "age" : 32,
                    "about" : "I like to collect rock albums",
                    "interests" : [
                        "music"
                    ],
                    "join_date" : "2017/06/01"
                }
            }
        ]
    }
}
```

Elasticsearch 会通过相关性来排序搜索结果。在第一个结果中，John Smith 的 about 字段明确地写到了 rock climbing。而在 Jane Smith 的 about 字段中则提到了 rock，但是并没有提及 climbing，同样作为匹配结果，但后者的 _score 就要比前者的低。

如果你想搜索的其实是“rock climbing”这个短语，可以使用 match_pharse 来查询，如代码清单 8-16 所示。

代码清单 8-16　短语搜索

```
$ curl -XGET 'localhost:9200/users/info/_search' -d '
{
    "query": {
        "match_phrase": {
            "about": "rock climbing"
        }
    }
}'
```

这时就只会返回用户 John Smith 的信息。

这里介绍的搜索内容，仅仅是浅尝辄止，考虑到篇幅原因，更多的搜索特性都未提及，如果你感兴趣，可以查阅相关文档。不过总体来说，Elasticsearch 确实很容易使用，不需要特别的配置，只要往 Elasticsearch 里添加数据就可以开始你的搜索之旅了。

8.5.3　节点和集群

一个运行 Elasticsearch 实例的服务器可称为一个节点（Node）。每个节点都有自己的

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

节点名称（node.name），默认情况下该名称是在启动时分配给节点的随机通用唯一标识符（UUID），也可以自定义节点名称。

集群（Cluster）由一个或者多个配置相同集群名称（cluster.name）的节点组成，默认情况下单个节点是只有一个节点的特殊集群。对用户来说，节点加入和退出集群非常简单，只需要修改节点的 cluster.name 配置即可。请注意，一个节点同一时间下只能加入一个集群，集群之间的名称不能相同。

1. 主节点和数据节点

集群中的一个节点会被选举为主节点（master node）。主节点是集群的最高统治者，负责集群相关的操作，例如创建或删除索引、增加或移除节点等。集群中负责数据处理的节点可称为数据节点，例如索引和搜索文档。一个集群中拥有的数据节点越多，处理数据的能力就越强。

默认配置下，节点是主节点候选节点，都有资格参与主节点选举，也都可以作为数据节点。

```
node.master: true
node.data: true
```

上述两个配置可在配置文件 elasticsearch.yml 中找到。如果不希望节点参与主节点选举，只需将 node.master 配置为 false。如果不希望节点作为数据节点，只需将节点的 node.data 配置为 false。对于大规模的集群，需要考虑分离出一些节点作为专用的主节点候选的节点。

2. 集群健康状态

集群健康状态有绿色、黄色或红色三种状态值。绿色代表一切正常，集群功能齐全；黄色代表所有的数据都是可用的，集群功能齐全，但是某些副本（可参考 8.5.4 节）没有被分配；红色则代表因为某些原因，某些数据不可用，但集群仍然部分可用（它仍然会响应部分搜索请求），但是需要尽快修复它，因为有数据丢失。

查询集群健康状态如代码清单 8-17 所示。

代码清单 8-17　查询集群健康状态

```
$ curl -XGET 'http://localhost:9200/_cluster/health?pretty=true'
{
    "cluster_name" : "elasticsearch",
    "status" : "green",
    "timed_out" : false,
    "number_of_nodes" : 1,
    "number_of_data_nodes" : 1,
    "active_primary_shards" : 0,
    "active_shards" : 0,
    "relocating_shards" : 0,
    "initializing_shards" : 0,
    "unassigned_shards" : 0,
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
    "delayed_unassigned_shards" : 0,
    "number_of_pending_tasks" : 0,
    "number_of_in_flight_fetch" : 0,
    "task_max_waiting_in_queue_millis" : 0,
    "active_shards_percent_as_number" : 100.0
}
```

该返回结果显示集群名为 elasticsearch 的集群状态是 green。

8.5.4 索引分片和索引副本

实际上，索引只是一个逻辑命名空间（logical namespace），它指向一个或多个的索引分片（Shards）。分片才是存储文档的实际物理区域。

索引副本（Replica），即为索引创建一份或多份副本拷贝。索引副本一方面提高了数据的可靠性；另一方面，在搜索索引时，可以在副本上并行进行，扩展了搜索量 / 吞吐量。

从索引副本的角度，分片分为主分片和副本分片。Elasticsearch 在分配分片到节点时遵循主分片与其副本分片不分配在同一节点的原则。这是因为如果分配在同一节点，一旦节点故障，主分片和副本分片都可能丢失无法找回，导致数据丢失。

主分片的数量在索引创建后就固定下来了，但是副本分片的数量可以随时改变。默认配置下创建一个索引，其索引的分片数为 5（number_of_shards=5），索引副本数量为 1（number_of_replicas=1），这个索引总共拥有的分片数为 10，即 5 个主分片 +5 个副本分片。

代码清单 8-18 创建了一个名为 jenkins 的索引，并将索引的分片数设置为 3。

代码清单 8-18 创建索引

```
curl -XPUT 'localhost:9200/jenkins' -d '
{
    "settings" : {
        "number_of_shards" : 3,
        "number_of_replicas": 1
    }
}'
```

在一个有三个节点的集群里，这个索引的分片的分配可能如图 8-5 所示。

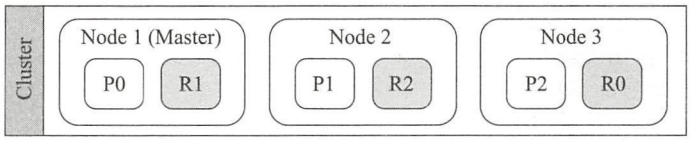


图 8-5 分片在节点中的分配

8.5.5 分布式特性

Elasticsearch 天生支持分布式。同时，它尽可能地屏蔽了分布式系统的复杂性，很多分

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

布式的操作都是由底层自动完成的，对用户完全透明，例如：

- 分配文档到分片中。
- 复制分片作为数据冗余备份。
- 按节点数来均衡分配分片。
- 当新增或删除一个节点时，集群都会重新平均分配所有的数据。
- 将集群中任一节点的请求路由到存有相关数据的节点。

即使完全不了解内部机制也不妨碍你使用 Elasticsearch，但是了解这些机制的细节将从另一个角度帮助你了解更完整的 Elasticsearch 知识。考虑到篇幅的原因，这些内部机制不在此展开，如果你感兴趣，可以翻阅相关的文档资料。

8.5.6　配置 Elasticsearch

以 Debian 系统为例，Elasticsearch 配置文件的默认配置目录为 /etc/elasticsearch。目录位置可以通过修改环境变量 ES_PATH_CONF 来修改配置目录，可以在 /etc/default/elasticsearch 中修改该变量。

本节开始就提到 Elasticsearch 提供了很好的默认设置，开箱即用。在实际的生产环境中，个别配置可能需要根据实际情况进行调整。

1. 路径配置

默认情况下，Elasticsearch 会把数据和日志放在安装目录下。如果使用默认路径，那么重装或升级 Elasticsearch 时，这些数据和日志被删除的风险很高。因此，在生产环境中，需要更改数据和日志文件夹的位置：

```
path.data: /var/data/elasticsearch
path.logs: /var/log/elasticsearch
```

path.data 设置可以设置为多个路径，在这种情况下，所有路径将用于存储数据：

```
path.data:
- /mnt/elasticsearch_1
- /mnt/elasticsearch_2
- /mnt/elasticsearch_3
```

以上配置存在与 elasticsearch.yml 中。

2. 集群名字和节点名字

在生产环境中，应该将这两个名称改为描述集群 / 节点用途的名称更为恰当。

```
cluster.name: jenkins-logging
node.name: jenkins_001_data
```

确保不要在不同的环境中使用相同的集群名称，否则会导致节点加入错误的集群。以上配置可在配置文件 elasticsearch.yml 中找到。

仅供非商业用途或交流学习使用

3. 最小主节点数

该配置可在配置文件 elasticsearch.yml 中找到。这个配置是为了防止脑裂。

集群主节点确定后，主节点会建立一个内部 ping 机制来监控集群内的其他节点是否在线。当集群中一个或多个节点与主节点失联，这些节点会重新选举出一个新的主节点，这样就出现一个集群同时存在两个主节点，这就是脑裂。脑裂会导致数据的丢失。配置最小主节点数就是告诉 Elasticsearch，当没有足够的主节点候选节点时，不要进行主节点选举，防止脑裂。

这个配置的数值必须设置为主节点候选节点个数的法定多数，即（主节点候选节点个数/2+1）。例如，集群中有 10 个主节点候选节点，那么法定多数就是 6。

```
discovery.zen.minimum_master_nodes: 6
```

4. 集群恢复相关配置

当集群重启时，集群恢复相关的配置项会影响分片恢复的表现。首先来了解下如果什么也不配置将会发生什么。

假设你的集群有 10 个节点，每个节点只保存一个分片。你进行了一次重启集群的操作，恰巧出现了 6 个节点已经启动，还有 4 个还没启动的场景。在线的 6 个节点相互通信，选出一个 master，从而形成一个集群。它们注意到数据不再均匀分布，因为有 4 个节点还在启动中，所以它们之间会立即启动分片复制。最后，其他 4 个节点启动完成加入了集群，这些节点会发现它们的数据正在被复制到其他节点，所以它们删除本地数据，然后整个集群重新进行平衡（因为集群的大小已经从 6 变成了 10）。在整个过程中，节点会消耗磁盘和网络带宽，来回移动数据。对于数据量很大的集群，这种无用的数据传输需要很长时间。如果等待所有的节点重启好了，整个集群再上线，那么所有的本地数据都不需要移动。

Elasticsearch 为了缓解这个问题，设置了一些配置。首先，设置了一个阈值：

```
gateway.recover_after_nodes: 8
```

这个配置的意思是：当集群中有多少个节点在线时，恢复将启动。

然后告诉 Elasticsearch 集群中应该有多少个节点，我们愿意为剩下的启动中的节点等待多久：

```
gateway.expected_nodes: 10
gateway.recover_after_time: 5m
```

这三个配置加在一起后意味着：

1）首先等待 8 个节点上线。

2）当条件 1 满足后，恢复将启动。等待集群节点到达 10 个或者等待 5 分钟，两个条件中任一满足，立即启动恢复。

以上配置可在配置文件 elasticsearch.yml 中找到。

5. 单播与组播

生产环境中，Elasticsearch 会配置为单播发现，而不是组播发现，以防止节点无意中加入集群。

```
discovery.zen.ping.multicast.enabled: false
```

使用单播，需要为 Elasticsearch 实例提供一些它可以去尝试连接的节点列表，例如：

```
discovery.zen.ping.unicast.hosts: ["host1", "host2:port"]
```

以上配置可在配置文件 elasticsearch.yml 中找到。

6. JVM 配置

通常很少需要更改 Java 虚拟机（JVM）选项。唯一最可能需要修改的是 heap size。有两种方式修改 heap size，最简单的是修改 ES_HEAP_SIZE 环境变量。服务进程在启动的时候会读取这个变量。

```
export ES_HEAP_SIZE=10g
```

还有就是在配置文件 /etc/default/elasticsearch 中加入如下语句：

```
ES_HEAP_SIZE=10g
```

Elasticsearch 默认安装后设置的 heap size 是 1GB。在实际的生产环境中，1GB 太小了，从性能角度考虑，需要将这个值改大，但不是越大越好，官方建议不要超过 32GB。

8.6　Kibana

Kibana 是一个专门为 Elasticsearch 服务的开源分析及可视化平台。Kibana 提供了 Web 界面，可以快速创建各种仪表板（dashboard）来搜索、查看存储在 Elasticsearch 中的数据。

8.6.1　让 Kibana 连接到 Elasticsearch

默认情况下，Kibana 会连接运行在本机上的 Elasticsearch。如需连接其他 Elasticsearch 实例，需修改 /etc/kibana/kibana.yml 里的 elasticsearch.url，然后重启服务。

```
# The URL of the Elasticsearch instance to use for all your queries.
# elasticsearch.url: "http://localhost:9200"
```

访问 localhost/status 可查看 Kibana 的状态，如图 8-6 所示。

如果使用的是默认配置，则通过浏览器访问 localhost:5601 即可见 Kibana Dashboard，如图 8-7 所示。

图 8-6　查看 Kibana 的状态

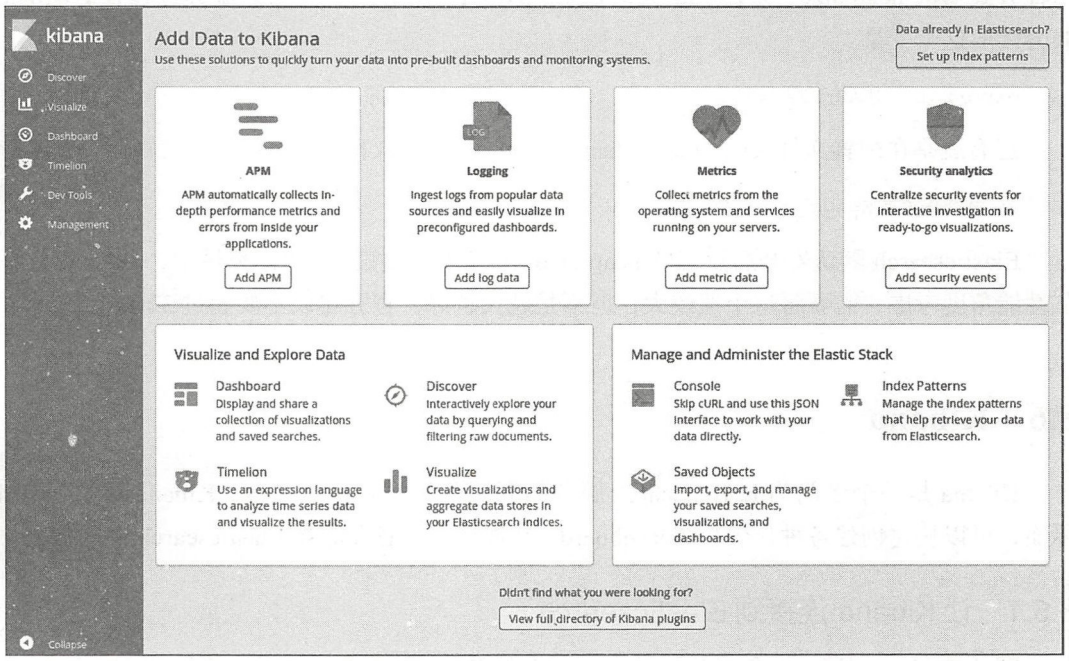


图 8-7　Kibana Dashboard

8.6.2　Index Pattern

第一次访问 Kibana 的时候，系统会提示你定义一个或多个 Index Pattern。请注意，如对接的 Elasticsearch 没有任何数据时，无法在界面上创建 Index Pattern。这里我们将一套莎士比亚的全部作品的数据导入到 Elasticsearch 中进行如下的学习[⊖]。

⊖ Loading sample data, https://www.elastic.co/guide/en/kibana/6.2/tutorial-load-dataset.html。

数据导入成功后，就能创建 Index Patterns 了。管理 Index Pattern 的位置在 Management-> Index Patterns 下。

1）在 Index Pattern 下输入 shakes*，这样就会匹配到 shakespeare 这个索引，如图 8-8 所示。

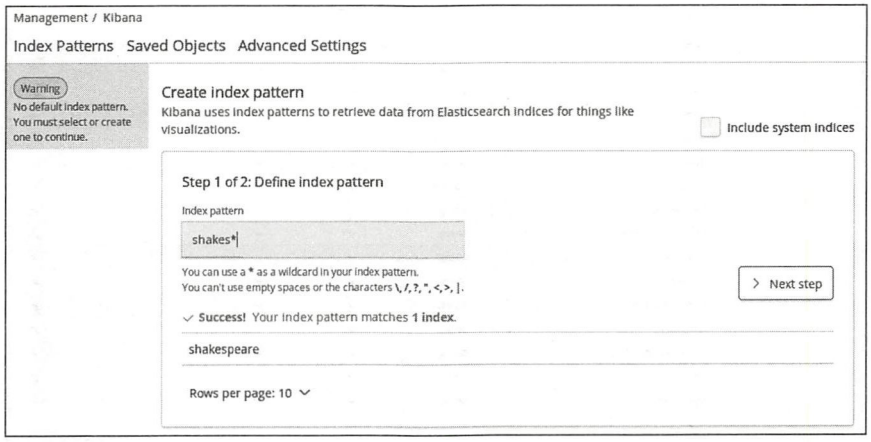


图 8-8　创建 Index Pattern（1）

2）点击下一步，进入 Index Pattern 的配置，如图 8-9 所示。

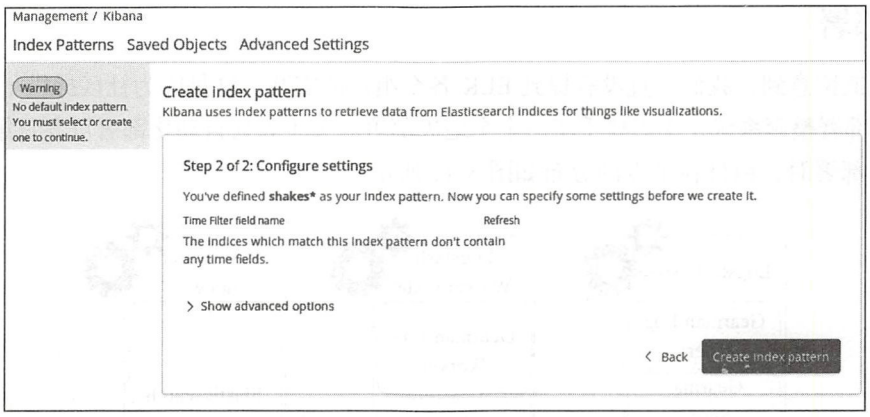


图 8-9　创建 Index Pattern（2）

3）点击 create index pattern 完成创建，页面显示如图 8-10 所示。

有了 Index Pattern 后，你可以：

- 在 Discover 页搜索和浏览你的数据；
- 在 Visualize 页转换数据成图表；
- 在 Dashboard 页创建定制自己的仪表板。

关于 Kibana 的使用请参考官方手册[⊖]。

⊖ Kibana 用户手册，https://www.elastic.co/guide/cn/kibana/current/index.html。

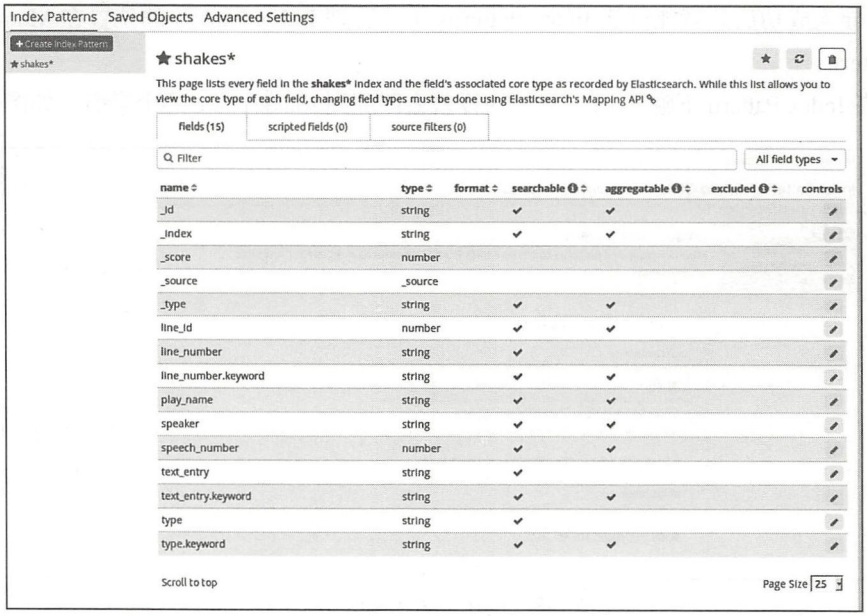


图 8-10　创建 Index Pattern（3）

8.7　部署

你可能留意到，我们一直没有提到 ELK 各个组件的安装，这是因为社区提供了自动化部署方法来部署整套系统，不需要手工一个个地去部署。关于如何自动化部署可参见第 10 章。

社区部署时，组件在节点的分布如图 8-11 所示。

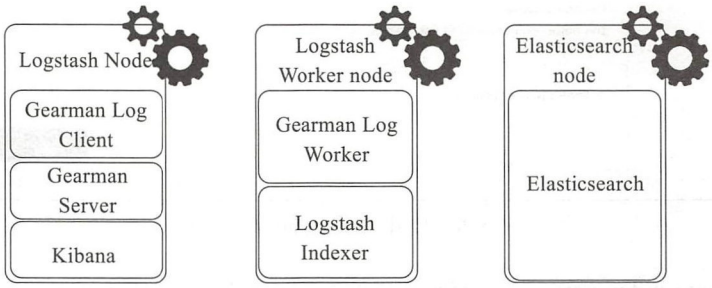
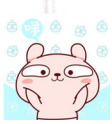


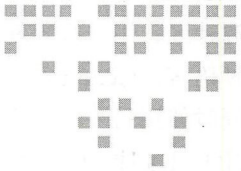
图 8-11　节点规划

部署完成后，需要登录到各个节点上去手动启动所部署的服务。

8.8　本章小结

本章介绍了日志分析系统的架构和使用到的 ELK Stack 中各个组件的基本知识点和简单的使用操作。





第 9 章 Chapter 9

公共组件详解

本章介绍 OpenStack CI/CD 系统中使用的三个公共组件：Gearman、ZeroMQ 和 ZooKeeper，它们在系统中的作用如下。

- Gearman。用于组件之间的通信，比如：Zuul 通过 Gearman 向 Jenkins 分发任务；Nodepool 可以通过 Gearman 从 Zuul 和 Jenkins 中获取 Slave 的使用情况等。
- ZeroMQ。Jenkins 通过 ZeroMQ 发布任务的启动、完成等事件，供 Nodepool、日志分析系统读取。
- ZooKeeper。主要用于协助 Nodepool 存储镜像制作信息。

9.1 任务分发系统（Gearman）

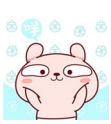
9.1.1 Gearman 介绍

为了扩展 Jenkins 的性能、实现 Jenkins 的 HA，以及在分布式系统的各个组件之间或者组件内部通信，在 CI/CD 系统中引入了 Gearman 组件。

Gearman 提供了一种通用的应用程序框架，将任务分发到适当的机器或者进程当中。它为应用程序提供了并行工作、负载均衡处理的能力，以及在不同编程语言之间调用的能力。Gearman 能够在很多应用程序中使用，比如高可用的 Web 站点、数据库事件复制分发系统等，换句话说，它就是一个负责分布式处理通信的中枢系统。

Gearman 的优点包括[⊖]如下部分。

⊖ http://gearman.org。



- 开源：Gearman 是自由软件，它遵循 BSD 协议，可以免费使用。Gearman 有一个活跃的开源社区，用户很容易就可以获得社区的支持。
- 支持多种语言：Gearman 支持多种编程语言接口。用户可以使用一种语言编写 Client 程序，同时使用另一种语言编写 Worker 程序。
- 灵活：用户不必拘泥于固定的设计模式，可以选择任何模型（例如 Map/Reduce 模型），从而快速地将分布式应用整合在一起。
- 快速：Gearman 的协议简单，接口经过了优化，支持线程化，服务端使用 C/C++ 实现，从而减小了应用的开销。
- 可植入：因为 Gearman 非常快速和轻量，从而适用于各种规模的应用程序。它可以很容易地被引入到现有的应用程序中，而且开销很小。
- 无单点故障：Gearman 不仅可以用来帮助扩展系统，同时还可以帮助系统实现容错。
- 消息大小无限制：Gearman 支持的单个消息长达 4GB。如果需要支持更大的消息，也没有问题，Gearman 支持对消息进行分块及组装。

9.1.2 Gearman 架构和工作原理

Gearman 的应用系统通常包括三个部分：Client（客户端）、Worker（工作端）和 Job Server（任务服务器）。其架构如图 9-1 所示。

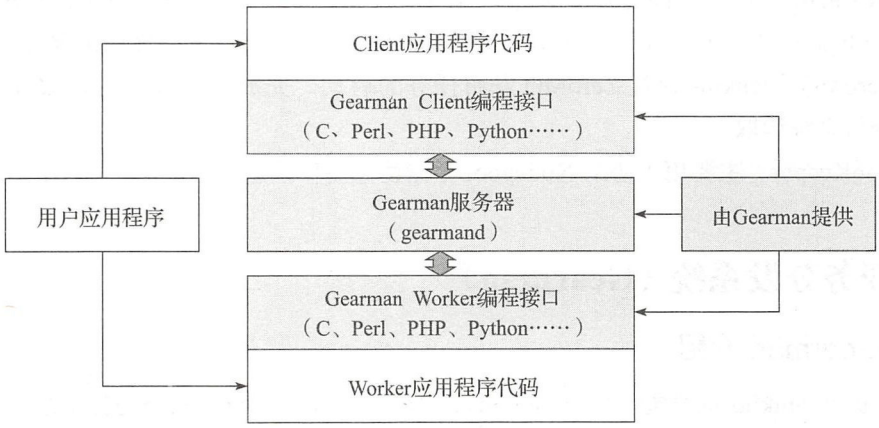


图 9-1 Gearman 应用系统架构

Gearman 提供了 Job Server（即 gearmand 程序，也可称为 Gearman 服务器或 Gearman Server），以及 Client 和 Worker 的 API。应用程序可以利用 Gearman 的 API 同 Job Server 通信，而不需要自己负责网络通信或者任务的映射关系。在 Gearman 内部，Client 和 Worker 的 API 通过 TCP 套接字与 Job Server 进行通信。

Client 的作用是创建 Job（任务），并把 Job 提交给 Job Server。Job Server 寻找一个能够运行这项 Job 的合适的 Worker，并把 Job 转发到这个 Worker 上来运行。Worker 执行 Client

请求的 Job，并且通过 Job Server 将响应消息返回给 Client。

1. Gearman 的典型用例

通过使用 Gearman，开发者可以在不同语言编写的 Client 和 Worker 之间进行通信。Gearman 支持的语言包括：C、C#/.NET、Go、Java、Lisp、Nodejs、PHP、Perl、Python、Ruby 等。

通过使用 Gearman，开发者可以将 Worker 代码放在更合适的机器（或一组机器）上执行，从而实现负荷均衡。举例来说：假设用户有一个 Web 应用程序需要进行图像格式转换，但是 Web 服务器的负荷比较重，用户就可以通过 Gearman 将图像发送到专门的一组机器上进行格式转换，这样就不会影响 Web 服务器的性能。

通过使用 Gearman，应用程序可以更好地利用多核服务器的性能，而且这种扩展非常简单。假设一台服务器上的 CPU 有 16 个核，用户就可以启动 16 个 Worker 程序的实例，每个实例绑定一个 CPU 核；不绑定的情况下，可以启动更多的 Worker 实例。

通过使用 Gearman，应用系统可以更简单地进行水平扩展。比如说，用户新购置了一组服务器，则只需要在它们上面安装和启动 Worker 程序，并连接到 JobServer 上，就可以实现系统性能的无缝扩展。

2. Gearman 集群

对于使用 Gearman 的系统来说，Job Server 是系统的中枢。用户可能担心 Job Server 会造成系统的单点故障。为了实现系统的高可用性，用户可以同时运行多个 Job Server，并让 Client 和 Worker 连接到第一个可用的 Job Server。这样，如果一个 Job Server 无法连接，Client 和 Worker 会自动转移到另一个 Job Server。图 9-2 显示了一个集群中有两个 Job Server 时的连接方式。

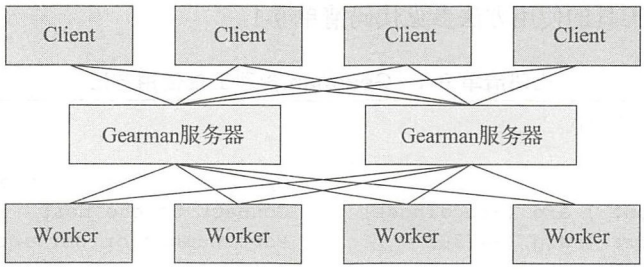


图 9-2　Gearman 集群

如果用户不希望运行太多 Job Server，那么运行 2 ～ 3 个即可实现高可靠性。Job Server 可以一次轻松地处理数百个 Client 和 Worker 的连接。在这种架构下，用户可以根据需要来设计自己的系统，扩展 Client 和 Worker，在容量允许的情况下，将工作负载分配到任意数量的机器上。

9.1.3　安装

使用 Gearman 服务，首先要安装 Gearman Job Server；为了查看服务的状态，通常还要

安装 Gearman 的管理工具；对于开发人员来说，还需要安装与开发语言对应的库，即 API。

1. Gearman Job Server

在 Ubuntu 操作系统上安装：

```
sudo apt-get install gearman-job-server
sudo service gearman-job-server start
```

除了包安装的方式外，用户也可以从 http://gearman.org/download/ 网站下载源码进行安装。

Gearman 服务使用的端口号默认为 4730。用户也可以在启动服务的时候使用 -p 或 --port 选项来指定端口：

```
sudo gearmand --daemon --port 5000
```

用户还可以将 Gearmand 所需的参数写入 /etc/gearmand.conf，格式与命令行参数相同，以避免每次启动服务时在命令行里输入参数。

在 OpenStack CI/CD 系统中，一些组件内置了 Gearman Job Server，例如 Zuul、日志分析系统等，不需要单独安装 Gearman Job Server。

2. Gearman 管理工具

用户可以安装 Gearman 管理工具来查询 Gearman 的状态。Gearman 管理工具与 Gearman Job Server 可以安装在不同的机器上。

在 Ubuntu 操作系统上安装：

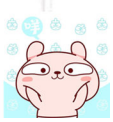
```
sudo apt-get install gearman-tools
```

Gearman 管理工具的使用方法参见代码清单 9-1。

代码清单 9-1　Gearman 管理工具使用方法

```
$ gearadmin --help
Options:
    --help                              Options related to the program.
    -h [ --host ] arg (=localhost)      Connect to the host
    -p [ --port ] arg (=4730)           Port number or service to use for con-
nection
    --server-version                    Fetch the version number for the server.
    --server-verbose                    Fetch the verbose setting for the server.
    --create-function arg               Create the function from the server.
    --drop-function arg                 Drop the function from the server.
    --getpid                            Get Process ID for the server.
    --status                            Status for the server.
    --workers                           Workers for the server.
    --shutdown                          Shutdown server.
```

Gearman 状态的查询方法参见代码清单 9-2。



代码清单 9-2　使用 Gearman 管理工具查询 Geaman 的状态

```
$ gearadmin --status
SayHello        0       0       1
.
```

该状态信息中，第一项为 Job 的名称，后面三个数字分别代表：等待运行的 Job 个数、正在运行的 Job 个数、能处理该 Job 的 Worker 的个数。

Worker 状态的查询方法参见代码清单 9-3。

代码清单 9-3　使用 Gearman 管理工具查询 Worker 的状态

```
$ gearadmin --workers
33 127.0.0.1 python-worker : SayHello
34 ::3438:3333:3000:0%4294967295 - :
.
```

Worker 状态信息中，包括了 Worker 的地址、ID 及其注册的 Job 名。同一台主机可以注册多个 Worker，同一个 Worker ID 可以注册多个 Job。

3. Gearman API

Gearman 支持多种编程语言，用户可以选择自己熟悉的语言编写 Worker 和 Client 程序。这里以 Python 语言为例进行说明。

首先安装 Gearman 的 Python 库：

```
sudo pip install gearman
```

编写一个 Worker 程序 worker.py。Worker 程序启动后，就会一直等待 Client 的请求；当它收到请求时，会返回一串字符，内容为“Hello”以及 Client 输入的信息。其内容参见代码清单 9-4。

代码清单 9-4　worker.py

```
import gearman

gm_worker = gearman.GearmanWorker(['localhost:4730'])

def task_sayhello(gearman_worker, gearman_job):
    print('Hello ' + gearman_job.data)
    return 'Hello ' + gearman_job.data

gm_worker.set_client_id('python-worker')
gm_worker.register_task('SayHello', task_sayhello)

gm_worker.work()
```

编写一个 Client 程序 client.py，它向 Job Server 发送一串字符，并显示从 Job Server 返回的字符。其内容参见代码清单 9-5。

非卖品，仅供非商业用途或交流学习使用

代码清单 9-5 client.py

```
import gearman

def check_request_status(job_request):
    if job_request.complete:
        print('Job %s finished! Result: %s - %s' %
              (job_request.job.unique, job_request.state, job_request.result))
    elif job_request.timed_out:
        print('Job %s timed out!' % job_request.unique)
    elif job_request.state == JOB_UNKNOWN:
        print('Job %s connection failed!' % job_request.unique)

gm_client = gearman.GearmanClient(['localhost:4730'])

my_job_request = gm_client.submit_job('SayHello', 'Tom')
check_request_status(my_job_request)
```

先在一个命令行窗口里运行 Worker 端程序，然后启动另一个命令行窗口运行 Client 端程序。Client 端的运行结果如代码清单 9-6 所示。

代码清单 9-6 client.py 运行结果

```
$ python client.py
Job 16a4ce54997de4354490ed69cbe8a8cd finished! Result: COMPLETE - Hello Tom
```

Worker 端的运行结果如代码清单 9-7 所示。

代码清单 9-7 worker.py 运行结果

```
$ python worker.py
Hello Tom
```

在 Worker 端程序运行前后，用户可以使用 Gearman 管理工具查询 Gearman 及其 Worker 的状态变化。从示例代码中可以看出，GearmanWorker 和 GearmanClient 实例初始化时指定的 Job Server 信息为列表形式，开发者可以在代码中指定多个 Job Server 的信息，即同时连接多个服务器。例如：

```
gm_worker = gearman.GearmanWorker(['server1', 'server2:5000'])
```

这里，server1 和 server2 为 Job Server 的域名，也可以填写它们的 IP 地址。其中，server1 的服务端口为默认值 4730，server2 的端口为自定义的端口 5000。

9.1.4 利用 Gearman 实现 Jenkins 的 HA

在 OpenStack CI/CD 系统中，Jenkins Master 本身不具有 HA 的能力，随着系统负荷的不断增加，Jenkins Master 会成为系统的瓶颈。在引入 Gearman 之前，系统的架构如图 9-3 所示[⊖]。

⊖ https://docs.openstack.org/infra/publications/gearman-plugin/。

非卖品，仅供非商业用途或交流学习使用

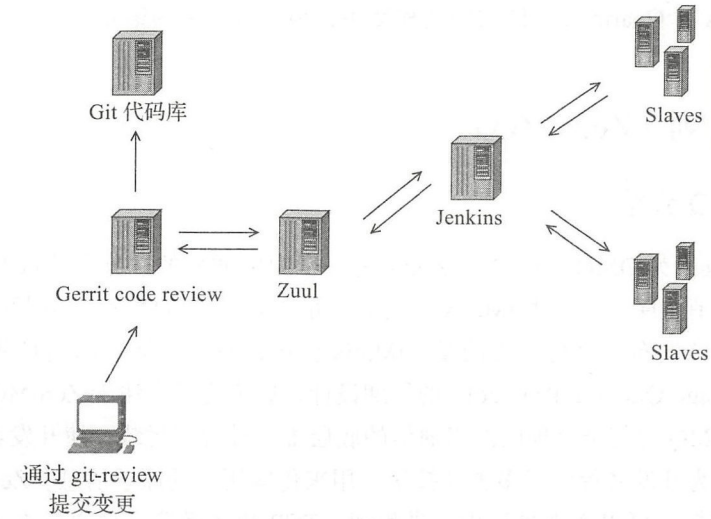


图 9-3　引入 Gearman 之前的系统架构

OpenStack CI/CD 系统引入 Gearman 之后，在 Jenkins Master 上安装 Gearman 插件，使其向 Gearman Job Server 注册 Worker，Worker 信息中包含了 Jenkins Slave 和 Job 的信息；Zuul 不再与 Jenkins 直接交互，而是提交执行任务的请求给 Gearman Job Server，由 Gearman Job Server 将任务分发给 Jenkins 注册的 Worker。系统中同时部署多个 Jenkins Master，不同的 Jenkins Master 在 Gearman Job Server 中注册的 Worker 可以提供相同任务名，Gearman Job Server 在分发任务时可以选择合适的 Worker 来处理任务，从而实现了 Jenkins Master 的 HA 和负荷分担。引入 Gearman 之后的系统架构如图 9-4 所示。

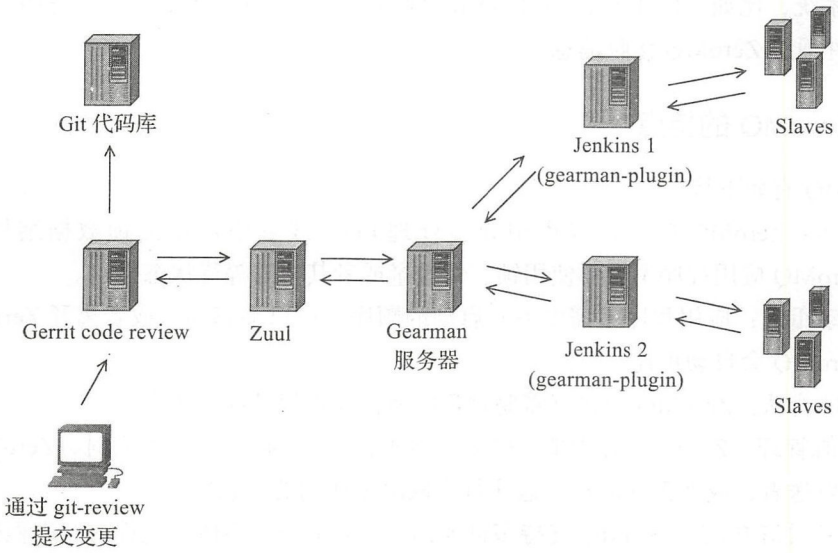


图 9-4　引入 Gearman 之后的系统架构

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

有关 Jenkins 的 Gearman 插件的安装和使用，请参考本书的第 4 章。

9.2 消息队列（ZeroMQ）

9.2.1 ZeroMQ 介绍

ZeroMQ（也称为 ØMQ、0MQ 或 zmq）是一个为可伸缩的、分布式或并发应用程序设计的高性能异步消息库。它是由 iMatix 公司和大量贡献者组成的社区共同开发的开源软件，以 LGPLv3 许可证发布。值得一提的是，iMatix 公司也参与了知名的消息队列协议 AMQP（Advanced Message Queuing Protocol）的早期设计，后来为了专注于 ZeroMQ 而宣布不再支持 AMQP。ZeroMQ 对复杂繁琐的网络通信的底层细节进行了封装，使开发者可以更加专注于上层应用。它为开发者提供了多种套接字，用来传输用户的原子消息。ZeroMQ 的套接字支持多对多的连接；可以实现进程内、进程间、TCP 和多播等多种通信方式；支持多种工作模式，如“请求 - 应答”“发布 - 订阅”“任务分发”等。使用者可以把它当作一个嵌入式网络消息开发库，也可以用它来构建一个并发框架。

ZeroMQ 为用户提供了一个消息队列，但与很多消息中间件不同的是，ZeroMQ 的运行不需要专门的消息代理（message broker）。它的名字中的 zero 代表了零代理、（尽可能）零延迟，也代表了一些其他目标：零管理、零开销、零浪费。与很多其他同类消息软件相比，ZeroMQ 的最大特点是使用非常简单，而且性能很高。

ZeroMQ 可以在大多数操作系统上运行，并支持多种开发语言，包括：C、C + +、C#、Go、Java、JavaScript、Perl、PHP、Python、Ruby 等。开发者可以同时使用多种编程语言，组成异构系统，比如：使用 C 语言开发的软件通过 ZeroMQ 发布消息，并使用 Python 语言开发的软件通过 ZeroMQ 接收消息。

9.2.2 ZeroMQ 的特点

ZeroMQ 有如下特点。

- ❑ 异步：ZeroMQ 在后台线程中异步处理 I/O。线程中使用无锁数据结构，并发的 ZeroMQ 应用程序不需要使用锁、信号量或者其他的等待状态指示。
- ❑ 自动重连：应用程序不需要关心启动的顺序，可以动态加入或者离开 ZeroMQ 网络，ZeroMQ 会自动重连。
- ❑ 智能排队：ZeroMQ 会在有需要时智能地自动将消息放进队列。
- ❑ 阈值管理：ZeroMQ 有阈值（称为“高水位”）机制。当队列满时，ZeroMQ 自动阻塞发送者，或者丢弃消息，这些行为取决于使用的消息模式。
- ❑ 多种通信方式：ZeroMQ 支持多种通信方式：TCP、多播、进程内、进程间，无须修改代码以使用不同的方式。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- 缓慢 / 受阻节点处理：ZeroMQ 会安全地处理速度较慢或受阻塞的节点，会根据消息模式使用不同的策略。
- 多种消息路由模式：ZeroMQ 提供了多种消息路由的模式，比如：请求 – 应答、发布 – 订阅。使用者可以根据自己的网络架构、拓扑来选择不同的模式。
- 增加代理：用户可以创建代理，来缓存、转发或者捕获消息。代理可以降低网络中互联的复杂度。
- 消息完整：ZeroMQ 会发送完整的消息，使用简单的帧格式来传递。接收端收到的消息与发送端发送的消息完全一致。
- 消息透明：ZeroMQ 对消息的格式没有要求，消息长度可以从 0 字节到千兆字节。
- 智能容错：ZeroMQ 会通过自动重试以智能地处理网络错误。
- 降低能耗：ZeroMQ 占用 CPU 开销很小，这意味着使用者可以节约能耗，降低碳排放。

9.2.3　ZeroMQ 的工作模式

ZeroMQ 提供了多种套接字用于网络连接。与普通的 TCP 连接相比，ZeroMQ 的连接有如下特点。

- 使用多种通信方式：inproc（进程内）、ipc（进程间）、tcp、pgm（广播）、epgm 等。
- 一个套接字可以有多个输出和输入连接。
- 没有提供类似 zmq_accept() 的函数，因为当套接字绑定到端点时它就自动开始接受连接。
- 网络连接在后台自动建立。当连接被破坏后 ZeroMQ 会自动重连（在对端恢复可达的情况下）。
- 应用程序的代码不能直接访问这些连接，因为它们被封装在 ZeroMQ 的套接字底层。

ZeroMQ 的套接字提供了多种消息模式，核心的消息模式包括：

- 请求 – 应答（Request-reply）模式：将一组服务端和一组客户端相连，用于远程过程调用或者任务分发。
- 发布 – 订阅（Pub-sub）模式：将一组发布者和一组订阅者相连，用于数据分发。
- 管道（Pipeline）模式：使用扇入或扇出的形式连接多个节点，可以有多个步骤或循环，用于并行任务分发和收集。
- 排他对（Exclusive pair）模式：将两个套接字一对一的连接起来，该连接是排它的。这种模式用于将一个进程内的两个线程连接起来。

以下是可用于连接 – 绑定对（connect-bind pair）的合法的套接字组合（任何一端都可以绑定）：

- PUB 和 SUB
- REQ 和 REP
- REQ 和 ROUTER（注意，REQ 会插入额外的空帧）

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- DEALER 和 REP（注意，REP 会假定有一个空帧）
- DEALER 和 ROUTER
- DEALER 和 DEALER
- ROUTER 和 ROUTER
- PUSH 和 PULL
- PAIR 和 PAIR

其中，PUB 套接字和 SUB 套接字用于“发布 - 订阅”模式；REQ 套接字和 REP 套接字用于“请求 - 应答”模式；PUSH 套接字和 PULL 套接字用于单向消息推送；PAIR 用于进程内通信。

对于一些复杂的场景，比如说我们需要让多个 Client（使用 REQ 套接字）和多个 Server（使用 REP 套接字）进行通信，如果让它们之间直接连接，一旦 Server 数目发生了变化，则所有的 Client 都需要进行修改。ZeroMQ 使用 DEALER 套接字和 ROUTER 套接字解决这种问题：REQ 套接字和 ROUTER 套接字通信，DEALER 套接字和 REP 套接字通信，ROUTER 和 DEALER 之间则需要进行消息转发。

本节对于 ZeroMQ 只是进行一般性的介绍，以及提供一些简单的例子。如果读者想深入了解 ZeroMQ 的知识，学习更多示例代码，而不限于其在 OpenStack CI/CD 中的应用，则请参阅 ZeroMQ 指导文档[⊖]。

9.2.4 安装

如前所述，ZeroMQ 不需要运行单独的消息代理，也不需要自己的服务。因此，安装 ZeroMQ 只需要安装开发运行所需的库即可。

如果使用 Python 语言开发 ZeroMQ 应用程序，则需要安装 pyzmq 包：

```
sudo pip install pyzmq
```

9.2.5 应用示例

本节中的示例程序均使用 Python 语言编写。

1. 请求 - 应答模式

本节介绍一个“请求 - 应答”模式的例子，包含客户端（REQ）和服务器端（REP）两个程序。服务端根据客户端请求消息的内容，应答不同的响应消息。其架构如图 9-5 所示。

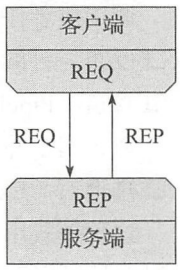


图 9-5 请求 - 应答模式

服务端程序 rep-server.py 的内容参见代码清单 9-8。

代码清单 9-8 rep-server.py

```
import time
```

⊖ http://zguide.zeromq.org。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
import zmq

context = zmq.Context()
socket = context.socket(zmq.REP)
socket.bind("tcp://*:5678")

data = {"name": "Tom", "age": 25}

while True:
    message = socket.recv()
    print("Received request: %s" % message)
    time.sleep(1)
    socket.send(str(data.get(message, "I don't understand")))
```

客户端程序 req-client.py 的内容参见代码清单 9-9。

代码清单 9-9　req-client.py

```
import zmq

context = zmq.Context()
socket = context.socket(zmq.REQ)
socket.connect("tcp://localhost:5678")

for request in ["name", "age", "address"]:
    print(">>> Sending request: %s ..." % request)
    socket.send(request)
    message = socket.recv()
    print("<<< Received reply: %s [ %s ]" % (request, message))
```

先在一个命令行窗口里运行服务端程序，然后启动另一个命令行窗口运行客户端程序。客户端的运行结果参见代码清单 9-10。

代码清单 9-10　req-client.py 运行结果

```
$ python req-client.py
Connecting to server ...
>>> Sending request: name ...
<<< Received reply: name [ Tom ]
>>> Sending request: age ...
<<< Received reply: age [ 25 ]
>>> Sending request: address ...
<<< Received reply: address [ I don't understand ]
```

服务端的运行参见代码清单 9-11。

代码清单 9-11　rep-server.py 运行结果

```
$ python rep-server.py
Received request: name
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
Received request: age
Received request: address
```

2. 发布 – 订阅模式

本节将介绍一个“发布 – 订阅”模式的例子，包含客户端（SUB）和服务器端（PUB）两个程序。服务端循环发送 1 ～ 10 这十个数字，客户端接收一个完整的循环，并打印累加结果。其架构如图 9-6 所示。

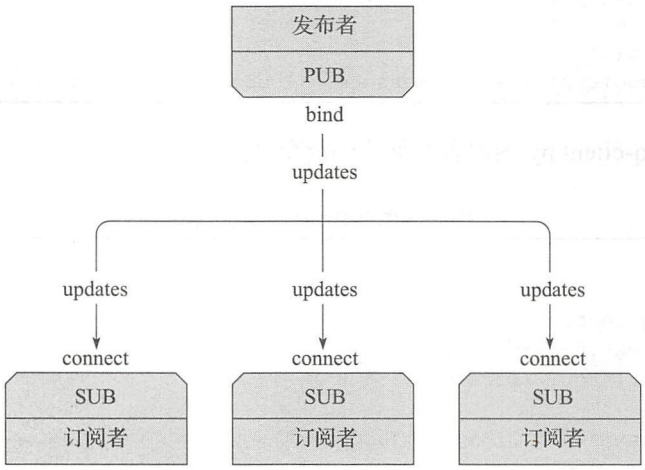


图 9-6　发布 – 订阅模式

服务端程序 publisher.py 的内容参见代码清单 9-12。

代码清单 9-12　publisher.py

```
import time
import zmq

context = zmq.Context()
socket = context.socket(zmq.PUB)
socket.bind("tcp://*:5678")

while True:
    socket.send("start")
    for num in range(10):
        socket.send(str(num + 1))
    socket.send("end")
    time.sleep(1)
```

客户端程序 subscriber.py 的内容参见代码清单 9-13。

代码清单 9-13　subscriber.py

```
import zmq
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
context = zmq.Context()
socket = context.socket(zmq.SUB)

socket.connect("tcp://localhost:5678")
socket.setsockopt(zmq.SUBSCRIBE, "")

start = False
total = 0

while True:
    data = socket.recv()
    if not start and data == "start":
        start = True
    elif data == "end":
        print "End. Total = ", total
        break
    elif data.isdigit():
        total += int(data)
```

先在一个命令行窗口里运行服务端程序，然后启动另一个命令行窗口运行客户端程序。服务端没有打印输出，客户端的运行结果参见代码清单 9-14。

代码清单 9-14　subscriber.py 运行结果

```
$ python subscriber.py
End. Total = 55
```

使用 SUB 套接字时，还可以在调用 socket.setsockopt() 接口时设置过滤器，从服务端发送的众多消息中过滤出自己感兴趣的内容，而不是全部接收，本例中没有设置过滤规则。

9.2.6　ZeroMQ 在 OpenStack CI/CD 系统中的作用

在 OpenStack CI/CD 系统中，ZeroMQ 主要用于 Jenkins 和 Nodepool、日志分析系统之间通信。Jenkins 通过 ZeroMQ 插件（ZMQ Event Publisher）使用“发布 – 订阅”模式将任务的启动、完成等事件发布出去，供 Nodepool、日志分析系统等组件消费。

关于 Jenkins 插件请参考第 4 章。

ZeroMQ 在 Nodepool 和日志分析系统中的使用请参考第 6 章和第 8 章。

9.3　分布式协调服务（ZooKeeper）

9.3.1　ZooKeeper 介绍

一个分布式系统，通常需要设计和开发一些协调服务，如果由应用设计和实现这些服务，需要执行额外的工作以解决竞争和死锁问题。ZooKeeper 的目的就是为分布式应用解决

仅供非商业用途或交流学习使用

这些问题，提供开箱即用的、可靠的、可扩展的、高性能的协调服务。

ZooKeeper 是一个面向分布式应用系统的开源协调服务，Google Chubby 的一个开源实现，Hadoop 的子项目。它提供了一套简单的服务原语，分布式应用基于它可以实现同步、配置管理、群组和名字服务。它运行在 Java 虚拟机上，提供 Java 和 C 的接口。ZooKeeper 设计为易于编程，为此它使用一种类似文件系统目录树风格的数据模型。

ZooKeeper 是实现复杂同步协调服务的基础，它提供如下保证。

- 顺序性：客户端的更新请求都会根据它发出的顺序被顺序地处理。
- 原子性：一个更新操作要么成功要么失败，没有其他可能的结果。
- 一致性：客户端不论连接到哪个服务器，展示给它都是同一个视图。
- 可靠性：一旦一个更新被应用就被持久化了，除非另一个客户端的更新请求修改了当前值。
- 实时性：对于每个客户端它的系统视图都是最新的。

9.3.2 ZooKeeper 架构和工作原理

ZooKeeper 本身也是一个分布式应用程序。ZooKeeper 遵循一个简单的“客户端 – 服务器”模型，其中客户端是使用服务的节点，而服务器是提供服务的节点。ZooKeeper 服务器的集合形成了一个 ZooKeeper 集合体（ensemble）。ZooKeeper 服务架构逻辑图如图 9-7 所示。

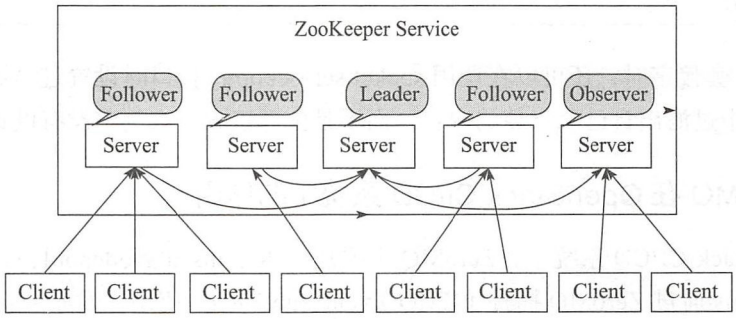


图 9-7 ZooKeeper 服务架构逻辑图

通常 ZooKeeper 由 2n + 1 台服务器组成，每个服务器都知道彼此的存在。每个服务器都维护一个内存状态镜像以及持久化存储的事务日志和快照。对于 2n + 1 台服务器，只要有 n + 1 台（大多数）服务器可用，整个系统就保持可用。

在任何给定的时间内，一个 ZooKeeper 客户端只连接到一个 ZooKeeper 服务器。每个 ZooKeeper 服务器都可以同时处理大量客户端连接。每个客户端定期发送 ping 到它所连接的 ZooKeeper 服务器，让服务器知道它处于活跃和连接状态。被 ping 的 ZooKeeper 服务器通过 ping Ack 进行响应，表示服务器也处于活跃状态。如果客户端在指定时间内没有收到服务器的确认响应，那么客户端会连接到集合体中的另一台服务器，而且客户端会话会被透

仅供非商业用途或交流学习使用

明地转移到新的 ZooKeeper 服务器。

1. 工作原理

ZooKeeper 服务器分为如下角色。

- 领导者（Leader）：负责进行投票的发起和决议，并最终更新系统状态。
- 跟随者（Follower）：用于接收客户端请求并返回结果给客户端，参与领导者发起的投票。
- 观察者（Observer）：可以接收客户端连接，将写请求转发给领导者。它不参与投票过程，只是同步领导者的状态。它的目的是为了扩展系统，提高读取速度。
- 学习者（Learner）：和领导者进行状态同步的服务器统称学习者，上述跟随者和观察者都是学习者。

系统启动时，集群中的服务器会选举出一台服务器作为领导者，其他的就作为跟随者（这里先不考虑观察者角色）。接着由跟随者来响应客户端的请求，对于不改变系统一致性状态的读操作，由跟随者直接给客户端返回存放在本地内存数据库中的数据；对于会改变系统状态的更新操作，则交由领导者进行提议和收集投票结果，票数超过半数则表示通过，跟随者返回结果给客户端。

ZooKeeper 的核心是原子广播，这个机制保证了各个服务器之间的同步。实现这个机制的协议叫作 Zab 协议。Zab 协议有两种模式，它们分别是恢复模式和广播模式。当服务启动、领导者崩溃或者领导者失去了大部分的跟随者支持时，Zab 就进入了恢复模式。恢复模式需要重新选举出一个新的领导者，当领导者被选举出来，且大多数服务器完成了和领导者的状态同步以后，恢复模式就结束了。状态同步保证了领导者和其他服务器具有相同的系统状态。

一旦领导者和多数的跟随者进行了状态同步，它就可以开始广播消息了，即进入广播状态。这时候当一个服务器加入 ZooKeeper 服务中，它会在恢复模式下启动，发现领导者，并和领导者进行状态同步。待到同步结束，它也参与消息广播。ZooKeeper 服务一直维持在广播状态，直到领导者崩溃了或者领导者失去了大部分的跟随者支持。广播模式类似于分布式事务中的两阶段提交：即领导者提起一个决议，由跟随者进行投票，领导者对投票结果进行计算并决定是否通过该决议，如果通过就执行该决议（事务），否则什么也不做。

广播模式需要保证决议按顺序处理，因此 ZooKeeper 采用了递增的事务 id 号（zxid）来保证。所有的提议都在被提出的时候加上了 zxid。实现中 zxid 是一个 64 位的数字，其中高 32 位是 epoch，用来标识领导者关系是否改变，每有一个领导者被选出来，它都会有一个新的 epoch，低 32 位是个递增计数。

2. 数据模型

ZooKeeper 拥有一个分层的命名空间，这个和标准的 Unix 文件系统非常相似，如图 9-8 所示。

仅供非商业用途或交流学习使用

从图 9-8 中，我们可以看出 ZooKeeper 有一个类似于 Unix 文件系统的数据模型，采用了树形层次结构，树中的每个节点都被称为 ZNode（ZooKeeper 数据节点）。可以将 ZNode 视为类似 Unix 传统系统中的文件，但它们可以有子节点；也可将它们视为目录，但它们可以有与其相关的数据。

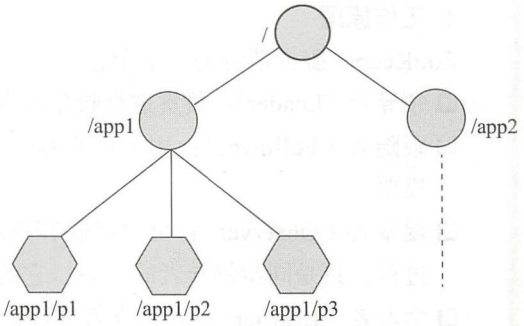


图 9-8 ZooKeeper 数据模型图

ZooKeeper 命名空间中每个节点使用路径来标识，如同 Unix 系统中的文件路径，必须用绝对路径，因此路径必须由“/”字符来开头。

ZooKeeper 设计用来存储协调数据（比如状态信息、配置、位置信息等），所以每个节点上关联的数据通常都比较小，一般几字节到几千字节。每个节点维护一个状态结构，包括数据变更版本、ACL（访问控制列表）变更版本、时间戳、数据长度等信息，每当节点关联的数据更新，它的数据变更版本号就加 1。

每个节点上存储的数据都要被原子性地读写操作，即读操作将获取与节点相关的所有数据，写操作将替换掉节点的所有数据，不能部分读取和修改。另外，每个节点都有自己的访问控制列表，这个列表限定了特定用户对目标节点可以执行的操作。

ZooKeeper 中的节点按生命周期管理分为临时节点和永久节点。

- 临时节点：该节点的生命周期依赖于创建它们的会话。一旦会话结束，临时节点将被自动删除，当然也可以手动删除。虽然每个临时的节点都会绑定到一个客户端会话，但它们对所有的客户端都是可见的。另外，临时节点不允许拥有子节点。
- 永久节点：该节点的生命周期不依赖于会话，并且只有在客户端显式执行删除操作的时候，它们才能被删除。

另外，ZooKeeper 还支持顺序节点（无论顺序还是临时或永久，都是节点的一种属性，节点可以有多种属性，比如同时有顺序和临时属性）。创建此类节点的时候，用户请求在节点的路径结尾添加一个递增的计数。这个计数对于此节点的父节点来说是唯一的，它的格式为”%10d”（10 位数字，没有数值的数位用 0 补充，例如”0 000 000 001”）。当计数值大于 2 147 483 647 时，计数器将溢出。

ZooKeeper 支持观察（watch）概念，即客户端可以在节点上设置观察，一旦此节点状态发生改变（比如节点数据变更，子节点增加、删除）时将触发观察对应的操作。当观察被触发时，ZooKeeper 将会向客户端发送且仅发送一条通知，因为观察只能被触发一次，这样可以减少网络流量。

3. 服务原语

ZooKeeper 提供了很简单的服务原语，即编程接口，包括以下操作：

- create：创建节点（父节点必须存在）。

仅供非商业用途或交流学习使用

- delete：删除节点（不能有子节点）。
- exists：测试节点是否存在，并获取它的元数据。
- getData：获取节点数据。
- setData：设置节点数据。
- getChildren：获取此节点下所有子节点列表。
- getACL：获取节点 ACL。
- setACL：设置节点 ACL。
- sync：等待数据同步到 ZooKeeper，使客户端的节点视图和 ZooKeeper 同步一致。

9.3.3　ZooKeeper 的安装和配置

1. 前提条件

ZooKeeper 运行在 Java 虚拟机上，需要在 ZooKeeper 运行的机器上安装 JDK（JDK6 或以上版本）。你可以使用包管理工具安装或从官网下载 Java JDK，Java JDK 的具体安装配置请参考其他资料，此处不再描述。

2. ZooKeeper 的安装

从 Apache 官网[⊖]下载一个 ZooKeeper 发布版本包。

将 tar 包下载到本地，解压到当前目录，示例如下：

```
tar xzvf zookeeper-3.4.11.tar.gz
```

3. ZooKeeper 的配置

ZooKeeper 有三种安装模式。

- 单机模式：单个主机上运行 ZooKeeper 服务进程，适用测试、开发环境。
- 集群模式：多个主机上运行 ZooKeeper 服务进程，组成一个服务集群，适用生产环境。
- 伪集群模式：单机上运行多个服务器进程实例模拟集群，适用于测试环境，这种模式实际较少使用，故本文不说明。

（1）ZooKeeper 单机模式配置

在 ZooKeeper 配置目录下创建或编辑 conf/zoo.cfg 文件，配置字段如代码清单 9-15 所示。

代码清单 9-15　ZooKeeper 单机模式配置字段

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
```

⊖ http://zookeeper.apache.org/releases.html。

- tickTime：基本事件单元，以毫秒为单位。它用来指示心跳，最小的会话过期时间为两倍的 tickTime。
- dataDir：内存数据库快照文件的存放位置，如果不设置特定参数，更新事务日志文件也被存储到此位置。
- clientPort：客户端连接的监听端口。

（2）ZooKeeper 集群模式配置

在 ZooKeeper 安装目录下创建或编辑 conf/zoo.cfg 文件，配置字段如代码清单 9-16 所示。

代码清单 9-16　ZooKeeper 集群模式配置字段

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
initLimit=5
syncLimit=2
server.1=zoo1:2888:3888
server.2=zoo2:2888:3888
server.3=zoo3:2888:3888
```

新增参数说明如下。

- initLimit：此配置表示允许跟随者服务器连接并同步到领导者服务器的初始化时间，它以 tickTime 的倍数来表示。当超过 tickTime*initLimit 时间，则连接失败。
- syncLimit：领导者服务器与跟随者服务器之间信息同步允许的最大时间间隔，如果超过此间隔，默认跟随者服务器与领导者服务器之间的链接断开。
- server.id：值格式为 host:adminPort:electionPort，它指示了不同的 ZooKeeper 服务器的自身标识，作为集群的一部分的机器应该知道集合体中的其他机器。ZooKeeper 服务器可以从 server.id 中读取相关信息。在服务器的 data 目录（dataDir 参数所指定的目录）下创建一个文件名为 myid 的文件，这个文件中仅含有一行内容，指定的是自身的 id 值。比如服务器“1”应该在 myid 文件中写入“1”。这个数值必须是集合体中唯一的，且大小在 1 到 255 之间。这一行配置中，管理端口（adminPort）是跟随者服务器连接到领导者服务器的端口，选举端口（electionPort）是用来进行领导者选举的端口。在这个例子中，每台机器的 ZooKeeper 服务器使用三个端口，分别是：clientPort：2181，adminPort：2888，electionPort：3888。

（3）ZooKeeper 服务启动

进入 ZooKeeper 安装目录下的 bin 子目录，执行如下脚本启动 ZooKeeper 服务：

```
sudo ./zkServer.sh start
```

单机模式下查看 ZooKeeper 服务状态的命令示例如代码清单 9-17 所示。

代码清单 9-17　ZooKeeper 服务状态查询命令

```
$ ./zkServer.sh status
JMX enabled by default
Using config: /etc/zookeeper/conf/zoo.cfg
Mode: standalone
```

查询命令的返回结果显示此 ZooKeeper 服务器配置为单机模式，正常运行。

集群模式下查看 ZooKeeper 服务状态的命令示例如代码清单 9-18 所示。

代码清单 9-18　ZooKeeper 服务状态查询命令

```
$ ./zkServer.sh status
JMX enabled by default
Using config: /etc/zookeeper/conf/zoo.cfg
Mode: leader
```

查询命令的返回结果显示此 ZooKeeper 服务器配置为集群模式，作为领导者角色正常运行。

9.3.4　ZooKeeper 典型应用

下面介绍使用 ZooKeeper 提供的服务原语来实现配置管理、分布式锁、群组这些典型应用，更多的应用场景实现请参考官网文档[⊖]。

1. 配置管理

分布式应用部署在多台机器上，一般都有公共配置，要逐个改变配置就变得很不方便。可以把这些配置信息作为某数据节点的数据保存到 ZooKeeper 中，数据节点树如图 9-9 所示。

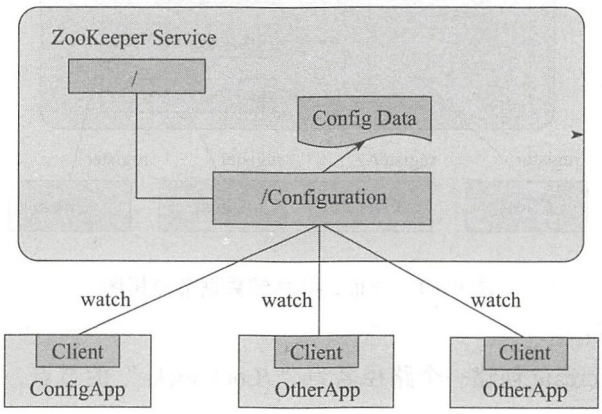


图 9-9　配置管理数据节点树图

⊖ http://zookeeper.apache.org/doc/current/recipes.html。

由一个客户端作为配置管理程序调用 create 创建这个配置节点和 setData 更新此节点上配置数据，其他客户端应用程序调用 getData 读取此节点上数据获取配置信息，并对这个目录节点进行监听，一旦配置信息发生变化，即此配置节点数据变更，每个客户端程序都会收到 ZooKeeper 的通知，然后客户端调用 getData 从 ZooKeeper 获取此节点新的配置信息应用到系统中，这就实现了分布式应用程序的集中配置管理。

2. 分布式锁

分布式锁服务可以分为两类。

- 保持独占：即锁同时只被一个用户占用，锁占用期间其他用户再申请锁则返回失败。实现流程如下：我们将 ZooKeeper 上的一个节点看作是一把锁，通过 create 节点的方式来实现。即所有客户端都去创建 /Lock 节点，由于 ZooKeeper 保证节点的唯一性，只有一个客户端可以创建此节点，最终成功创建的那个客户端即拥有了这把锁。用完后删除掉自己创建的 /Lock 节点就释放出锁。
- 控制时序：即申请锁的用户按发送请求的前后顺序依次可以获取到锁。实现方式如下：预先创建父节点 /Lock，所有客户端在它下面创建临时顺序编号子节点，编号最小的获得锁，用完后删除，依次处理，所有客户端可以按序获取，数据节点树如图 9-10 所示。

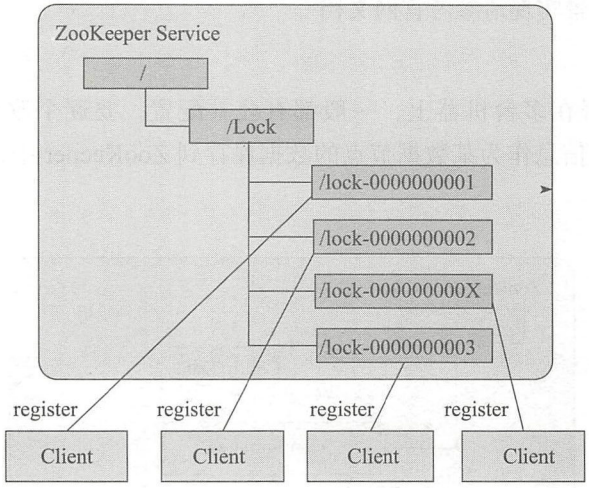


图 9-10　分布式共享锁数据节点树图

获取锁流程如下：

1）客户端调用 create 创建一个路径名为“/Lock/lock-”的节点，并将此节点设置为顺序和临时属性；

2）调用 getChildren 获取 /Lock 父目录节点下所有子节点信息，不设置观察点；

3）如果步骤 1 中创建的节点名中序号后缀数值是所有子目录下最小值，此客户端就获

取锁，退出流程；

4）否则客户端调用 exists 查询 /Lock 父目录下比本节点名中序号后缀数值小 1 的节点，同时设置观察点；

5）如果 exists 返回 false，跳转到步骤 2；否则等待步骤 4 设置的观察点触发的对应节点删除通知。

释放锁流程如下：

客户端主动删除步骤 1 中创建的锁节点或者客户端与 ZooKeeper 服务器的连接断开，其所创建的锁节点被自动删除。

3. 群组

组群管理主要包括检测组成员的加入和退出、选举群主。数据节点树如图 9-11 所示。

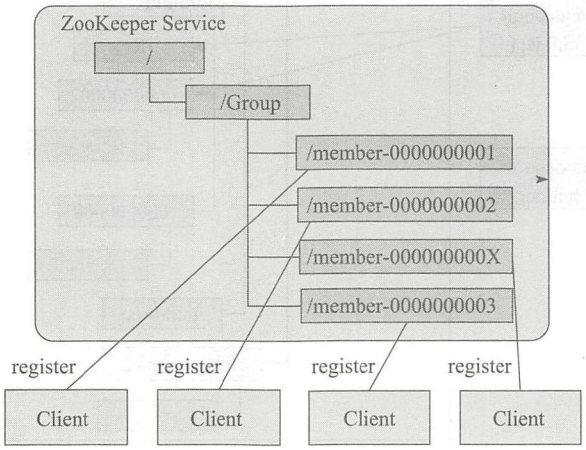


图 9-11　组群数据节点树图

加入组群流程：

1）客户端在一个父目录“Group”下创建一个临时和顺序属性的节点“member-x”（x 表示自动分配的序号）；

2）监听父节点，调用 getChildren 可以获取父目录节点下所有兄弟节点信息；

3）如果有新的兄弟节点被创建，所有其他客户端收到有新的成员加入组群的通知。

退出组群流程：

一旦有客户端与 ZooKeeper 服务器的连接断开，其所创建的临时节点被自动删除，或者客户端主动删除本节点，所有其他应用程序收到此客户端退出组群的通知。

选举群主流程：

参照上面的分布式锁获取流程，客户端都申请锁，成功获取锁的客户端成为群主，即编号最小的节点选举为群主。一旦群主异常导致和 ZooKeeper 的连接断开，或群主主动释放锁放弃群主角色，其他客户端重新开始以上流程选出新的群主。

9.3.5 Nodepool 中使用 ZooKeeper 示例

Nodepool builder 模块会定期构建系统镜像，并把构建的镜像信息保存到 ZooKeeper 中。Node- pool 使用 ZooKeeper 保存镜像信息的数据节点树图如图 9-12 所示。

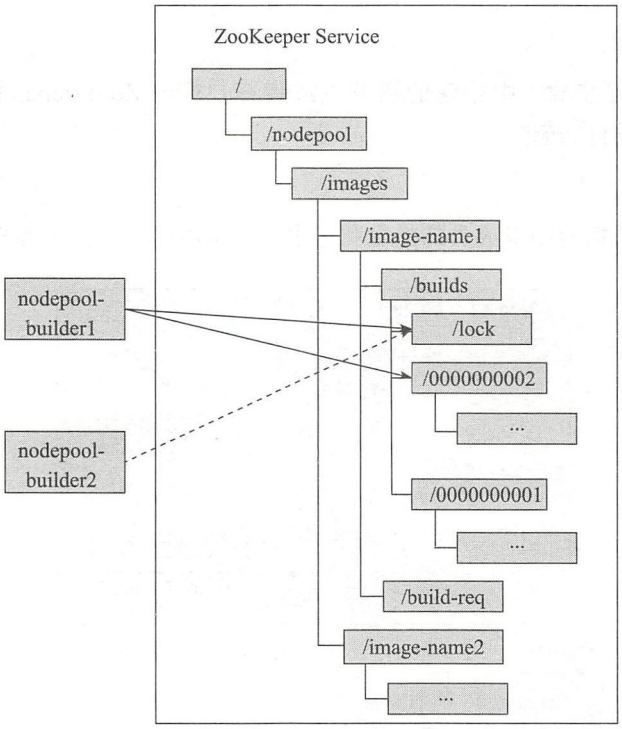


图 9-12 Nodepool 使用 ZooKeeper 保存镜像信息的数据节点树图

Builder 使用 ZooKeeper 保存构建的镜像信息大致流程如下：

1）Builder 创建 /nodepool/images 目录节点，读取配置文件获取需要构建的各镜像信息，在 /nodepool/images 节点下以镜像名作为节点名创建各镜像节点，如 /nodepool/images/ubuntu-trusty，在镜像目录节点下再创建 builds 子目录节点；

2）在某镜像节点的 builds 子目录节点下申请独占锁（参见前节分布式锁说明）。如果获取锁成功，在 builds 子目录节点下创建镜像构建节点，节点属性为顺序节点，这样可以保证最大数值的节点名对应节点是最新的镜像构建节点，同时将构建的镜像格式、构建时间、节点状态（值为 building）等信息写入此节点数据，当镜像构建命令执行完毕后，将节点状态更新为 ready，重新写入此节点数据；如果获取锁失败，则继续后面步骤；

3）对其他镜像节点继续执行步骤 2，直到所有镜像节点都执行完以上操作。

通过 ZooKeeper 自带的客户端脚本程序连接到 ZooKeeper 服务器可以查看相应信息，示例如代码清单 9-19 所示。

非卖品，仅供非商业用途或交流学习使用

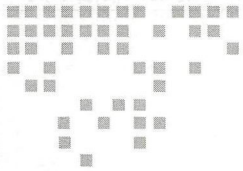
代码清单 9-19　Nodepool 在 ZooKeeper 中保存的镜像信息示例

```
# 进入 ZooKeeper 包安装目录下 bin 目录
$ cd /usr/share/zookeeper/bin

# 通过 ZooKeeper 客户端程序登录本机的 ZooKeeper 服务
$ ./zkCli.sh -server 127.0.0.1:2181
Connecting to 127.0.0.1:2181
Welcome to ZooKeeper!
JLine support is enabled
[zk: 127.0.0.1:2181(CONNECTING) 0]
WATCHER::

WatchedEvent state:SyncConnected type:None path:null

# 查看 ubunt-trusty 这个镜像的构建目录节点下子节点列表
[zk: 127.0.0.1:2181(CONNECTED) 7] ls /nodepool/images/ubuntu-trusty/builds
[lock, 0000000036, 0000000024]

# 查看 ubuntu-trusty 这个镜像对应的 0000000036 构建子节点的数据，building 表示此镜像在构建中
[zk: 127.0.0.1:2181(CONNECTED) 10] get /nodepool/images/ubuntu-trusty/builds/
0000000036
{"state": "building", "formats": "qcow2", "builder": "nodepool.cibook.oz",
"state_time": 1516475486.587287}

# 查看 ubuntu-trusty 这个镜像对应的 0000000024 构建子节点的数据，ready 表示此镜像构建完成
[zk: 127.0.0.1:2181(CONNECTED) 11] get /nodepool/images/ubuntu-trusty/builds/
0000000024
{"state": "ready", "formats": "qcow2", "builder": "nodepool.cibook.oz",
# 退出 ZooKeeper 客户端程序
[zk: 127.0.0.1:2181(CONNECTED) 15] quit

Quitting...
```

9.4　本章小结

本章对 Gearman、ZeroMQ 和 ZooKeeper 进行了简单介绍，并通过一些简单的示例代码来说明如何使用它们。如果读者希望对它们有进一步的深入了解，请参考相关资料。

非卖品，仅供非商业用途或交流学习使用



Chapter 10 第 10 章

社区 CI/CD 实践

前面的章节详细介绍了 OpenStack CI/CD 的框架及各组件的使用，本章将基于这套解决方案重点介绍下企业内部的实践：首先介绍如何利用社区的配置搭建一套单节点的系统，有个总体直观印象；然后介绍如何利用该系统定制企业级 CI/CD。

10.1 Puppet 简介

在介绍实践之前，我们先简单介绍一下 Puppet 配置管理工具。

Puppet 是由 Puppetlabs 公司开发的系统管理框架和工具集，于 2005 年正式面世，广泛应用于 IT 服务的自动化管理。由于良好的声明式语言和易于扩展的框架设计以及可重用可共享的模块，使得许多大型知名公司（比如 Google、Cisco、Twitter、RedHat 等）和机构在其数据中心的自动化管理中用到了 Puppet。2012 年 5 月，Puppetlabs 率先与 OpenStack 合作，使 Puppet 在 OpenStack 中发挥着重要的作用：OpenStack 基础设施团队将其用于 Wiki、Storyboard 任务跟踪、CI 系统等的运维管理中；社区提供了多个代码库用于 OpenStack 各服务的自动化部署和管理，如 puppet-nova、puppet-neutron、puppet-cinder 等；此外，Cisco、RedHat、Mirantis 等多家公司的 OpenStack 发行版或部署工具中均使用到了 Puppet。可以说，不了解 Puppet 就无法深入理解 OpenStack 的基础设施及服务的自动化部署管理。

10.1.1 概述

Puppet 是一种 Linux、Unix、Windows 平台的集中配置管理工具，使用自有的 Puppet 描述语言，可管理配置文件、用户、cron 任务、软件包、系统服务等。Puppet 把这些系统实体

非卖品，仅供非商业用途或交流学习使用

称之为资源，每种资源有不同的属性，因此 Puppet 语言就是描述这些资源的属性以及资源之间关系的语言。Puppet 的设计目标是简化对这些资源的管理以及妥善处理资源间的依赖关系。

Puppet 与其他手工操作工具有一个最大的区别就是 Puppet 的配置具有稳定性，它会让你的系统状态同配置文件所要求的状态保持一致，因此你可以多次执行 Puppet。一旦你更新了配置文件，Puppet 就会根据配置文件来更改你的资源状态。

首先了解下几个重要概念，有助于读者理解 Puppet 的工作模式。

1. 资源

定义目标状态的核心组件；核心资源包括：package、group、user、file、exec、cron、service 等。代码清单 10-1 定义了一个文件资源，指定该文件为 /etc/passwd，归属的用户和组为 root，没有则创建。

代码清单 10-1　文件资源

```
file { '/etc/passwd':
    owner => 'root',
    group => 'root',
    ensure => present,
}
```

2. 类

一组资源的集合。如代码清单 10-2 所示：定义一个 Linux 类，包含 file 和 package 两个资源。

代码清单 10-2　Linux 类

```
class linux {
    file { '/etc/passwd':
        owner => 'root',
        group => 'root',
        ensure => present,
    }
    package {'httpd':
        ensure => installed,
    }
}
```

3. 模块

以资源为核心，是类的集合，如 module1、module2。一个模块下通常包括如下目录和文件：manifests、files、templates 和 metadata.json。代码清单 10-3 表示一个模块的目录结构。

代码清单 10-3　模块

```
$ tree
.
├── manifests
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
|   ├── implementation/
|   ├── init.pp
|   └── other_class.pp
├── files/
├── lib/
├── templates/
├── metadata.json
├── README.md
├── spec/
└── tests/
```

- manifests/：定义并保存资源，必须要包括一个 init.pp 的文件，这是该模块的初始（入口）文件，导入一个模块的时候，会从 init.pp 开始执行。可以把所有的代码都写到 init.pp 里面，也可以分成多个 pp 文件，init.pp 再去包含其他文件。
- files/：静态文件目录，可以通过访问 puppet:///modules/module_name/file 获取文件内容。
- templates/：模板文件目录。
- metadata.json：该模块的元数据。

4. 节点

puppet 利用模块 + 节点的方式，实现目标状态的定义；节点的管理一般存贮在文件 /etc/puppet/manifests/site.pp 里。如代码清单 10-4 所示，定义一个域名为 www1.example.com 的节点，引用两个模块 common 和 apache的资源状态。

代码清单 10-4　节点

```
node 'www1.example.com' {
    include common
    include apache
}
```

5. Manifest

清单，用于定义并保存资源，是一个资源组织工具；

6. Facter[⊖]

收集节点信息。代码清单 10-5 显示通过 facter 命令获取到的当前主机信息。

代码清单 10-5　Facter

```
$ facter
architecture => amd64
augeasversion => 1.4.0
blockdevice_sda_model => TOSHIBA MQ01ACF0
blockdevice_sda_size => 500107862016
blockdevice_sda_vendor => ATA
blockdevices => sda
```

⊖ https://docs.puppet.com/facter/。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
dhcp_servers => {"system"=>"10.67.14.37", "enp0s25"=>"10.67.14.37", "wlp3s0"=>
"192.168.1.1"}
domain => zte.intra
facterversion => 2.4.6
filesystems => ext2,ext3,ext4,squashfs,vfat
fqdn => opnfv.zte.intra
gid => zte
hardwareisa => x86_64
hardwaremodel => x86_64
hostname => opnfv
id => zte
...
```

7. Hiera

Manifest 同时包含静态代码和动态数据，动态数据需要根据具体情况赋值，因此将动态数据从 Manifest 中剥离出来，使用 Hiera 将数据存储在外部的文件中，根据实际情况赋值。在编译时，Manifest 通过 Hiera 动态查询所需的值，然后加载到伪代码（catalog）中。比如代码清单 10-6 表示在编译 jenkins.cibook.oz 的 catalog 时，需要通过 Hiera 动态查询配置数据。

代码清单 10-6　使用 Hiera

```
node 'jenkins.cibook.oz' {

    class { '::openstackci::jenkins_node':
        vhost_name                  => $vhost_name_jenkins,
        project_config_repo         => hiera('project_config_repo'),
        serveradmin                 => hiera('serveradmin', "webmaster@${vhost_
name_jenkins}"),
        jenkins_version             => hiera('jenkins_version', 'present'),
        jenkins_vhost_name          => hiera('jenkins_vhost_name', 'jenkins'),
        jenkins_username            => hiera('jenkins_username', 'jenkins'),
        jenkins_password            => hiera('jenkins_password', 'XXX'),
        jenkins_ssh_private_key     => hiera('jenkins_ssh_private_key'),
        jenkins_ssh_public_key      => hiera('jenkins_ssh_public_key'),
        log_server                  => hiera('log_server'),
        # <...>
    }
}
```

而 Hiera 从哪些文件中读取这些动态配置数据，需要通过 Hiera 的配置文件指定，即配置 /etc/puppet/hiera.yaml，如代码清单 10-7 所示，表示从目录 /etc/puppet/environments 下查找 common 文件，文件类型为 yaml。

代码清单 10-7　Hiera 配置文件

```
---
:backends:
    - yaml
:logger: console
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
:hierarchy:
    - common

:yaml:
    :datadir: /etc/puppet/environments
```

具体参数详解参见官方文档[⊖]。

10.1.2 基础架构

Puppet 支持两种工作模式：Master/Agent 和 Stand-alone 模式，下面简单介绍一下两种模式的工作方式。

1. Master/Agent

Master/Agent 即客户端和服务器交互模式，通过 Master 集中式管理。Puppet master 上只运行 Puppetmaster 服务，简称 Puppet 服务器端；Puppet agent 上只运行 Puppet 服务，简称 Puppet 客户端。

Puppet 客户端首先会连接到 Puppet.服务器端，并且通过 Facter 工具获取客户端的基本配置信息并发送给服务器端。服务器端通过分析客户端的主机名，通过 node 定义，找到该主机的配置代码，然后编译配置代码为 catalog，把编译好的 catalog 发回客户端，客户端执行代码完成配置，并且把代码执行情况反馈给 Puppet 服务器端。

流程图如图 10-1 所示。

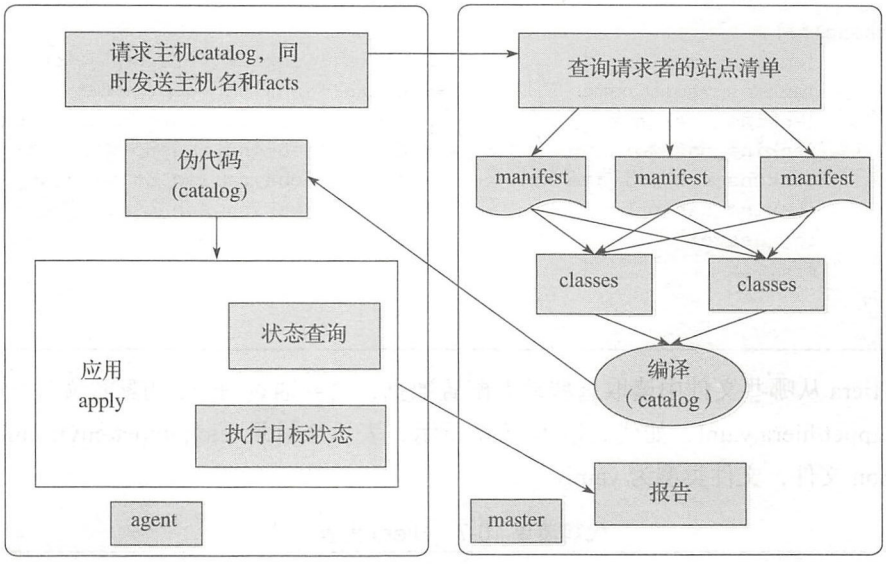


图 10-1 Master/Agent 工作模式

⊖ https://docs.puppet.com/hiera/。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

这种模式下的客户端和服务器端是通过 ssl 连接来保证安全性的，ssl 连接用到的证书是 x.509 证书。Puppet 会自己生成证书。ssl 在连接的时候，会对比证书里面的域名字是否和主机名匹配，如果不匹配就会报证书验证错误。

下面列举了几个 Master/Agent 模式的特点：

- 集中式管理，易于更新配置、上报和保存数据。
- 基于 ssl 的连接，安全性更好。
- Puppet agent 代理不编译自己的 catalog，从而为其指定的任务提供更多容量。
- 所有的配置和 Puppet 模块安装都在 Master 节点上，对性能要求较高，需要一个专用的服务器。
- 需要良好的网络连接。

在生产环境中，一般都采用 Master/Agent 模式。

2. Stand-alone

Stand-alone 即单机模式，只运行 Puppet 服务，收集配置信息、编译、执行配置代码和上报执行结果都在本机完成。

流程图如图 10-2 所示。

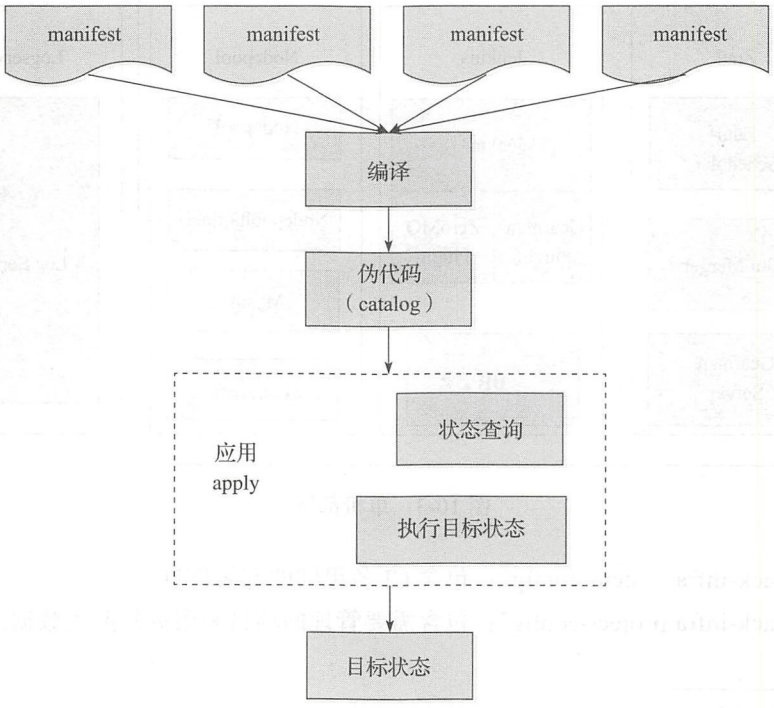


图 10-2　Stand-alone 工作模式

总体来说，Stand-alone 模式也可以完成 Master/Agent 模式的相同的功能。用户选择何

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

种模式工作，可以根据需要在安全性和其他方面做一些权衡。

本节只简单介绍了 Puppet 的工作原理，其他详细信息可以到官网上学习[㊀]。

10.2 单机部署

要把这样一个多组件的复杂的 CI 系统手动部署和配置起来是比较困难的，而且手动部署和配置还会增加人为误操作的风险，降低部署成功率。社区提供了一个自动化的单机[㊁]部署配置指导，框图如图 10-3 所示，即使用 Puppet Stand-alone 的工作模式，把 Zuul、Jenkins 和 Nodepool 部署安装在一台服务器上，日志服务器部署在另一台服务器上。

我们先从简单的入手，搭建这样一套 CI/CD 系统。

10.2.1 前期准备

1）在 GitHub 上 fork 两个代码库。

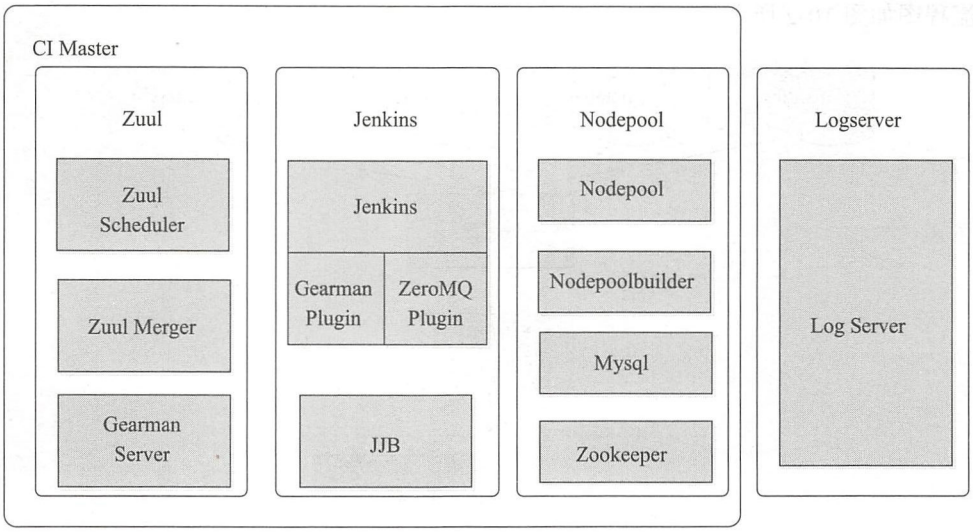


图 10-3 单机部署

- openstack-infra/system-config[㊂]：包含 CI 各组件的部署信息。
- openstack-infra/project-config[㊃]：包含需要管理的项目和服务的配置数据。

㊀ https://puppet.com/。
㊁ https://docs.openstack.org/infra/openstackci。
㊂ https://git.openstack.org/openstack-infra/system-config。
㊃ https://git.openstack.org/openstack-infra/project-config。

仅供非商业用途或交流学习使用

注解　openstack-infra/project-config 不能用主分支，要使用 before-jenkins-config-removed 标签分支。

2）可用的 OpenStack 环境：用于 Nodepool 创建节点资源池，需要创建 flavor、租户、租户网络及外部网络。

3）两台服务器：可以是虚机，也可以是物理机，8 核 8G 内存 80G 磁盘，一台作为日志服务器，一台作为 CI Master。

4）Gerrit 服务器：使用现有的 Gerrit 服务器，注册 CI 用户，用户设置中添加公钥用于 Zuul 通过该用户连接 Gerrit，并赋予该用户接收事件流和合并代码的权限。

10.2.2　安装部署

1. 安装和配置 Puppet

（1）安装 Puppet

克隆 system-config 库到 CI Master/ 日志服务器上，运行 system-config/install_puppet.sh 脚本自动安装。

（2）安装 Puppet 模块

运行脚本 system-config/install_modules.sh 安装所需的 Puppet 模块。具体需要安装哪些模块，是在 system-config/modules.env 文件中指定，这里我们暂时不做裁剪，采用社区的默认配置。这一步耗时比较长，因为在线下载的模块比较多。安装完成后，查看 /etc/puppet/modules/ 目录，可看到安装了哪些模块。

（3）配置“Puppet Stand-alone”工作模式

准备节点配置文件 site.pp，部署代码中有动态配置数据，还需要准备数据配置文件 common.yaml 和 Hiera 配置文件 hiera.yaml。

相关文件在项目 puppet-openstackci[⊖]的 contrib 目录下，该项目是 Puppet 的 openstackci 模块，已在上一步中安装。

❑ 对于搭建日志服务器，如代码清单 10-8 所示，复制文件到环境的配置目录下：

代码清单 10-8　日志服务器配置文件

```
cp /etc/puppet/modules/openstackci/contrib/hiera.yaml /etc/puppet
cp /etc/puppet/modules/openstackci/contrib/log_server_site.pp \
    /etc/puppet/manifests/site.pp
cp /etc/puppet/modules/openstackci/contrib/log_server_data.yaml \
    /etc/puppet/environments/common.yaml
```

❑ 对于搭建 CI Master 服务器，如代码清单 10-9 所示，复制文件到环境的配置目录下：

⊖ https://git.openstack.org/openstack-infra/puppet-openstackci。

仅供非商业用途或交流学习使用

代码清单 10-9 CI Master 服务器配置文件

```
cp /etc/puppet/modules/openstackci/contrib/hiera.yaml /etc/puppet
cp /etc/puppet/modules/openstackci/contrib/single_node_ci_site.pp \
    /etc/puppet/manifests/site.pp
cp /etc/puppet/modules/openstackci/contrib/single_node_ci_data.yaml \
    /etc/puppet/environments/common.yaml
```

节点配置文件 site.pp 和动态数据配置文件 common.yaml 是自动化部署的重点配置，下面是具体节点安装的详细介绍。

2. 日志服务器搭建

（1）动态数据配置

日志服务器的动态数据配置 common.yaml 比较简单，如代码清单 10-10 所示，带 # 的参数表示不是必须配置的。

代码清单 10-10 日志服务器动态配置

```
# See parameter documetation inside ../manifests/single_node_ci.pp
# Fields commented out have reasonable default values
domain: .oz
jenkins_ssh_public_key: your-jenkins-public-key-no-whitespace
#swift_authurl:
#swift_user:
#swift_key:
#swift_tenant_name:
#swift_region_name:
#swift_default_container:
```

- domain：日志服务器的域名信息。
- jenkins_ssh_public_key：用于 Jenkins 访问日志服务器的公钥。

其他 Swift 相关的配置是可选项，用于日志文件的对象存储，这些可以不配置。

（2）节点配置

日志服务器的部署入口文件 site.pp 如代码清单 10-11 所示。

代码清单 10-11 日志服务器节点配置

```
node default {
    class { '::openstackci::logserver':
        domain                => hiera('domain'),
        jenkins_ssh_key       => hiera('jenkins_ssh_public_key'),
        swift_authurl         => hiera('swift_authurl', ''),
        swift_user            => hiera('swift_user', ''),
        swift_key             => hiera('swift_key', ''),
        swift_tenant_name     => hiera('swift_tenant_name', ''),
        swift_region_name     => hiera('swift_region_name', ''),
        swift_default_container => hiera('swift_default_container', ''),
```

仅供非商业用途或交流学习使用

```
    }
}
```

调用 openstackci 模块中的 logserver 类完成该节点上的所有资源部署。具体执行哪些资源操作可查看 puppet-openstackci/manifests/logserver.pp，这里不再详细展开。

（3）安装部署

配置完成后运行如下命令执行安装部署：

```
sudo puppet apply /etc/puppet/manifests/site.pp
```

该命令执行时，通过 hiera 获取 common.yaml 文件中的动态配置数据，加载到该节点的清单，即 openstackci 模块中的 logserver 类，编译成 catalog 执行并上报结果，用户可以根据终端输出查看执行结果。

部署成功后，使用如下命令验证服务器是否可用：

```
scp -i $JENKINS_SSH_PRIVATE_KEY_FILE -o StrictHostKeyChecking=no \
    $your-log-file jenkins@<fqdn_or_ip>:/srv/static/logs/
```

JENKINS_SSH_PRIVATE_KEY_FILE 即与配置文件中公钥配对的私钥文件，your-log-file 是上传的日志文件。

文件上传完成之后，通过访问如下地址查看上传文件：

```
http://<fqdn_or_ip>/$your-log-file
```

3. CI Master 服务器搭建

（1）配置动态数据

CI Master上动态数据配置 common.yaml 如代码清单 10-12 所示。

代码清单 10-12　CI 服务器动态配置文件

```
# See parameter documentation inside ../manifests/single_node_ci.pp
# Fields commented out have reasonable default values

#vhost_name:
project_config_repo: https://github.com/dwj/project-config
#serveradmin:
jenkins_username: jenkins
jenkins_password: jenkins
jenkins_ssh_private_key: your-jenkins-private-key
jenkins_ssh_public_key: your-jenkins-public-key-no-whitespace
#java_args_override:
jjb_git_revision: 1.6.2
#jjb_git_url: https://git.openstack.org/openstack-infra/jenkins-job-builder
gerrit_server: gerrit.oz
#gerrit_ssh_host_key:
gerrit_user: openzeroci
```

仅供非商业用途或交流学习使用

```
gerrit_user_ssh_private_key: gerrit-user-private-key
gerrit_user_ssh_public_key: gerrit-user-public-key-no-whitespace
git_email: openzeroci@zte.com.cn
git_name: openzeroci
log_server: logs.oz
#smtp_host:
#smtp_default_from:
#smtp_default_to:
zuul_revision: 2.5.2
zuul_git_source_repo: https://git.openstack.org/openstack-infra/zuul
oscc_file_contents: |
                    # Do Not Edit - Generated & Managed by Puppet
                    #
mysql_root_password: root123
mysql_nodepool_password: root123
nodepool_jenkins_target: jenkins1
jenkins_api_key:         # jenkins api key get from Jenkins
jenkins_credentials_id: # jenkins api key get from Jenkins
nodepool_revision: 0.4.0
nodepool_git_source_repo: https://git.openstack.org/openstack-infra/nodepool
```

为了便于理解，对所有的配置参数按组件进行划分，并分别介绍：

(a) 公共配置

即所有的组件服务都需要用到的参数配置，具体包括：

- project_conftg_repo：project-config 仓库的地址，所有 CI 组件都需要读取该仓库下和自己相关的配置，即准备工作中 fork 的仓库地址。
- vhost_name：主机名称，可以不配。

(b) Zuul 相关配置

- gerrit_server：Zuul 对接的 Gerrit 服务器地址。
- gerrit_ssh_host_key：Gerrit 服务器上的公钥，用于 Zuul 通过 ssh 第一次连接 Gerrit 服务器时不需要人机交互。
- gerrit_user：Gerrit 用户，Zuul 连接 Gerrit 时使用的用户，需要提前在 Gerrit 服务器上创建。
- gerrit_user_ssh_private_key 和 gerrit_user_ssh_public_key：Gerrit 用户的密钥对，Zuul 通过 ssh 连接到 Gerrit 服务器时通过该私钥鉴权；公钥必须和 Gerrit 用户下配置的公钥一致。
- git_email 和 git_name：操作 git 库的用户和邮件地址。
- smtp_host、smtp_default_from 和 smtp_default_to：SMTP 相关的配置，Zuul 在测试完成后通过 SMTP 发送邮件通知，可以不配置。
- zuul_revision 和 zuul_git_source_repo：Zuul 的版本及源仓库地址，Puppet 安装 Zuul 时即从该仓库地址下载对应的版本安装。

- log_server：日志服务器的地址，即上节搭建的日志服务器的地址，Zuul 在返回当前变更的测试结果时携带本次测试的日志存放路径。

（c）Jenkins 相关配置

- serveradmin：可以不配置，默认值为 webmaster@${vhost_name}。
- jenkins_username 和 jenkins_password：访问 Jenkins 服务器的用户名和密码；如果不是 Jenkins 默认的管理员用户，在 Jenkins 安装完成之后，需要手动创建该用户和密码。
- jenkins_ssh_private_key 和 jenkins_ssh_public_key：Jenkins 服务中使用到的密钥对，与上节搭建日志服务器使用的密钥对相同，私钥用于访问日志服务器以及创建 credentials，公钥写入 Jenkins 的工作目录下。
- java_args_override：启动 Jenkins 时重写的参数选项，可以不配。
- jjb_git_revision 和 jjb_git_url：JJB 的版本及源仓库地址；可以不配，直接使用默认值 https://git.openstack.org/openstack-infra/jenkins-job-builder。

（d）Nodepool 相关配置

- oscc_ftle_contents：Nodepool 对接的云环境，参考配置如代码清单 10-13 所示，根据实际云环境修改。

代码清单 10-13　云环境定义

```
client:
    force_ipv4: true
clouds:
    zte-pod13:
        profile: internap
        floating_ip_source: 'neutron'
        auth:
            username: 'ciuser'
            password: 'ciuser'
            project_name: 'ciuser'
            auth_url: 'http://h13.oz:5000/v2.0'
        regions:
            - name: RegionOne
              values:
                  networks:
              - name: admin_floating_net
                  routes_externally: true
              - name: net-ci
                  nat_destination: true
                  default_interface: true
```

- mysql_root_password 和 mysql_nodepool_password：root 用户和 nodepool 用户访问数据库的密码，数据库安装完成之后会自动创建。
- nodepool_jenkins_target：和 Nodepool 对接的 Jenkins 名称。

- nodepool_revision 和 nodepool_git_source_repo：Nodepool 的版本及源仓库地址；Puppet 安装 Nodepool 时即从该仓库地址下载对应的版本安装。
- jenkins_api_key 和 jenkins_credentials_id：这两个参数暂且不配，待 Jenkins 服务器起来之后，需要手动配置并获取填入，后面讲 Jenkins 的手动配置时会提到，用于 Nodepool 访问 Jenkins 服务器。

（2）节点配置

CI Master的部署入口文件 site.pp 如代码清单 10-14 所示。

代码清单 10-14　CI Master 节点配置

```
node default {
# If the fqdn is not resolvable, use its ip address
$vhost_name = hiera('vhost_name', $::fqdn)

class { '::openstackci::single_node_ci':
    vhost_name                  => $vhost_name,
    project_config_repo         => hiera('project_config_repo'),
    serveradmin                 => hiera('serveradmin', "webmaster@${vhost_name}"),
    jenkins_version             => hiera('jenkins_version', 'present'),
    jenkins_vhost_name          => hiera('jenkins_vhost_name', 'jenkins'),
    jenkins_username            => hiera('jenkins_username', 'jenkins'),
    jenkins_password            => hiera('jenkins_password', 'XXX'),
    jenkins_ssh_private_key     => hiera('jenkins_ssh_private_key'),
    jenkins_ssh_public_key      => hiera('jenkins_ssh_public_key'),
    java_args_override          => hiera('java_args_override', undef),
    gerrit_server               => hiera('gerrit_server', 'review.openstack.org'),
    gerrit_user                 => hiera('gerrit_user'),
    gerrit_user_ssh_public_key  => hiera('gerrit_user_ssh_public_key'),
    gerrit_user_ssh_private_key => hiera('gerrit_user_ssh_private_key'),
    gerrit_ssh_host_key         => hiera('gerrit_ssh_host_key', ''),
    git_email                   => hiera('git_email'),
    git_name                    => hiera('git_name'),
    log_server                  => hiera('log_server'),
    smtp_host                   => hiera('smtp_host', 'localhost'),
    smtp_default_from           => hiera('smtp_default_from', "zuul@${vhost_name}"),
    smtp_default_to             => hiera('smtp_default_to', "zuul.reports@${vhost-_,
!name}"),
    zuulv2                      => hiera('zuulv2', true),
    zuul_revision               => hiera('zuul_revision', 'master'),
    zuul_git_source_repo        => hiera('zuul_git_source_repo','https://git.
openstack.org/openstack-infra/zuul'),
    oscc_file_contents          => hiera('oscc_file_contents', ''),
    mysql_root_password         => hiera('mysql_root_password'),
    mysql_nodepool_password     => hiera('mysql_nodepool_password'),
    nodepool_jenkins_target     => hiera('nodepool_jenkins_target', 'jenkins1'),
    jenkins_api_key             => hiera('jenkins_api_key', 'XXX'),
    jenkins_credentials_id      => hiera('jenkins_credentials_id', 'XXX'),
    nodepool_revision           => hiera('nodepool_revision', 'master'),
```

```
        nodepool_git_source_repo     => hiera('nodepool_git_source_repo','https://
git.openstack.org/openstack-infra/nodepool'),
        jjb_git_revision             => hiera('jjb_git_revision', '1.6.2'),
        jjb_git_url                  => hiera('jjb_git_url','https://git.openstack.
org/openstack-infra/jenkins-job-builder'),
    }
```

与日志服务器类似，调用 openstackci 模块中的 single_node_ci 类完成该节点上的所有资源部署。Puppet 模块的调用关系如图 10-4 所示。

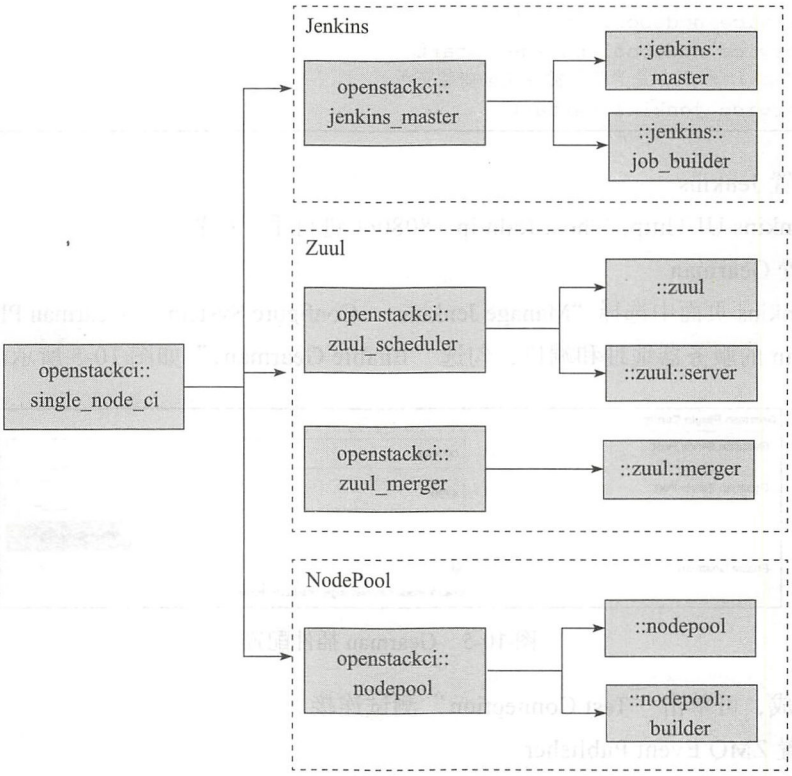


图 10-4　单机部署模块调用关系

具体执行哪些资源操作可查看 puppet-openstackci/manifests/single_node_ci.pp 进行了解，这里不做详细展开。

（3）安装部署

运行部署之前，需要在 /etc/hosts 文件中添加 jenkins 为 Apache 服务使用：

```
127.0.0.1 localhost jenkins
```

运行如下命令执行安装部署：

```
puppet apply /etc/puppet/manifests/site.pp
```

执行完成后所有组件服务基本都安装完成，但是系统还不能用，需要手工启动服务和配置 Jenkins。

（4）启动服务

运行代码清单 10-15 所示命令启动服务。

代码清单 10-15　服务启动

```
sudo service apache2 restart
sudo service zuul start
sudo service zuul-merger start
sudo service nodepool start
sudo service nodepool-builder start
# 重启 Jenkins 服务是为了使插件完全安装可用
sudo service jenkins restart
```

（5）配置 Jenkins

打开 Jenkins UI（http://<host fqdn/ip>:8080/）进行手工配置。

（a）配置 Gearman

进入 Jenkins 页面中选择“Manage Jenkins → Configure System → Gearman Plugin Config”，配置 Gearman 的服务器地址和端口，勾选“Enable Gearman，”如图 10-5 所示。

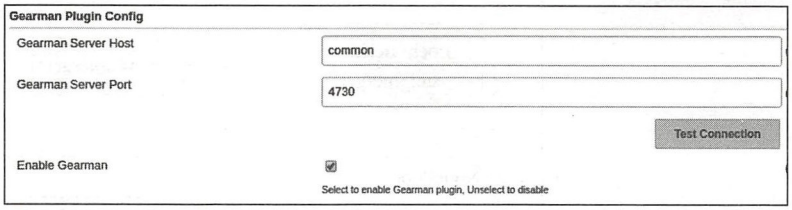


图 10-5　Gearman 插件配置

配置完成，可单击“Test Connection”测试连接。

（b）配置 ZMQ Event Publisher

进入 Jenkins 页面，选择 Manage Jenkins → Configure System → ZMQ Event Publisher，勾选“Enable on all Jobs”，如图 10-6 所示。

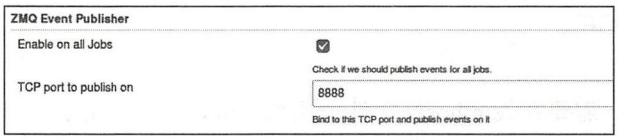


图 10-6　ZMQ 插件配置

（c）安全配置

默认情况下，Jenkins 安装时禁用了用户安全认证，对于外部访问受限的开发环境问题

不大，但还是强烈建议打开。

1）创建用户和密码：进入 Jenkins 页面选择“Manage Jenkins → Manage Users → Create User”，填入用户名和密码，和动态配置文件中提到的 jenkins_username 和 jenkins_password 保持一致。

2）获取用户 API token：进入 Jenkins 页面选择“Manage Jenkins → Manage Users → jenkins 用户→ Con-figure → Show API token”，将获取到的这串数字保存到动态数据配置文件中的 jenkins_api_key 中，Nodepool 使用该配置参数访问 Jenkins 的接口。

3）创建安全证书：进入 Jenkins 页面选择“Credentials → System → Global credentials → Add Creden-tials”，类型选择“ SSH Username with private key”，依次填写用户名“ jenkins”，私钥地址“/var/lib/jenkins/.ssh/id_rsa,”最后单击“OK”保存。

Jenkins 使用此证书来访问其他节点。证书创建完成后会产生一个“uuid”，将该“uuid”保存到动态数据配置文件中的 jenkins_credentials_id。

最后再运行如下命令使新配置生效。

```
puppet apply /etc/puppet/manifests/site.pp
```

至此，CI 系统搭建完成，但是如何应用起来，和已有的 Gerrit 服务器中的项目对接，还需要进行项目配置。

4. 项目配置

假设 Gerrit 上已有 openzero/sample-project 项目，以此为例介绍下如何配置 project-config 仓库来管理该项目的 CI 测试。这里不再对该仓库下的所有配置进行详细解释，只提供一个项目示例的最简配置。

project-config 代码目录结构如代码清单 10-16 所示：

代码清单 10-16　项目配置仓库目录结构

```
$ tree
.
├── jenkins
│   ├── jobs
│   │   ├── macros.yaml
│   │   ├── projects.yaml
│   │   ├── sample-project
│   │   │   └── sample-project.yml
│   │   └── test.yaml
│   └── views
│   └── test.yaml
├── nodepool
│   ├── elements/
│   ├── nodepool.yaml
│   └── scripts/
├── test-requirements.txt
```

```
├── tools/
├── tox.ini
├── view-requirements.txt
└── zuul
    ├── layout.yaml
    ├── openstack_functions.py
    └── zuul.conf-sample
```

首先要配置 Nodepool，使之与云环境对接，提供虚机挂载到 Jenkins 上；然后配置项目 Job，使用 JJB 翻译并上传到 Jenkins 服务器上；最后，在 Zuul 配置中增加该项目，并指定在什么 pipeline 下执行该 Job。

（1）Nodepool 目录

- elements：镜像编译需要用到的 elements。
- scripts：虚机起来之后，挂载到 Jenkins 之前需要运行的一些测试及配置脚本。这两项可暂不修改，采用社区默认配置。
- nodepool.yaml：Nodepool 的配置文件。

代码清单 10-17 是一个最简配置，其中 providers 段中 cloud 的配置必须和 common.yaml 中 oscc_file_contents 定义的云名称一致，具体配置参见第 6.4 节。

代码清单 10-17　Nodepool 配置

```
elements-dir: /etc/nodepool/elements
images-dir: /opt/nodepool_dib

cron:
    cleanup: '*/1 * * * *'
    check: '*/15 * * * *'

zmq-publishers:
    - tcp://localhost:8888

gearman-servers:
    - host: localhost

zookeeper-servers:
    - host: localhost
      port: 2181

labels:
    - name: ubuntu-trusty
      image: ubuntu-trusty
      ready-script: configure_mirror.sh
      min-ready: 3
      providers:
        - name: zte-pod13-RegionOne
providers:
```

```
    - name: zte-pod13-RegionOne
        region-name: 'RegionOne'
        cloud: zte-pod13
        api-timeout: 60
        boot-timeout: 120
        max-servers: 16
        rate: 0.001
        ipv6-preferred: false
        image-type: qcow2
        template-hostname: 'ci-book-template-{image.name}-{timestamp}'
        clean-floating-ips: true
        images:
        - name: ubuntu-trusty
            min-ram: 2048
            diskimage: ubuntu-trusty
            username: jenkins
            private-key: /home/nodepool/.ssh/id_rsa
            config-drive: true

targets:
    - name: jenkins1
        assign-via-gearman: True
        hostname: ci-book-jenkins-{label.name}-{provider.name}-{node_id}

diskimages:
    - name: ubuntu-trusty
        rebuild-age: 864000
        elements:
        - ubuntu-minimal
        - vm
        - infra-package-needs
        - initialize-urandom
        - jenkins-slave
        - simple-init
        - cache-devstack
        - openstack-repos
        - nodepool-base
        - prepare-node
        - zuul-worker
        - growroot
        - stackviz
    release: trusty
    env-vars:
        TMPDIR: /opt/dib_tmp
        DIB_CHECKSUM: '1'
        DIB_IMAGE_CACHE: /opt/dib_cache
        DIB_APT_LOCAL_CACHE: '0'
        DIB_DISABLE_APT_CLEANUP: '1'
        DIB_GRUB_TIMEOUT: '0'
        DIB_DEBIAN_COMPONENTS: 'main,universe'
        NODEPOOL_SCRIPTDIR: /etc/nodepool/scripts
```

运行如下命令使新配置生效。

```
puppet apply /etc/puppet/manifests/site.pp
```

Nodepool 开始编译镜像，这一步耗时比较久，至少需要 4 小时，完成后上传至云环境。用户可登录云环境查看镜像文件是否上传成功，如图 10-7 所示，上传了一个名为“template-ubuntu-trusty-1524061372”的镜像，包含镜像详细信息，以及 Nodepool 的定制属性。

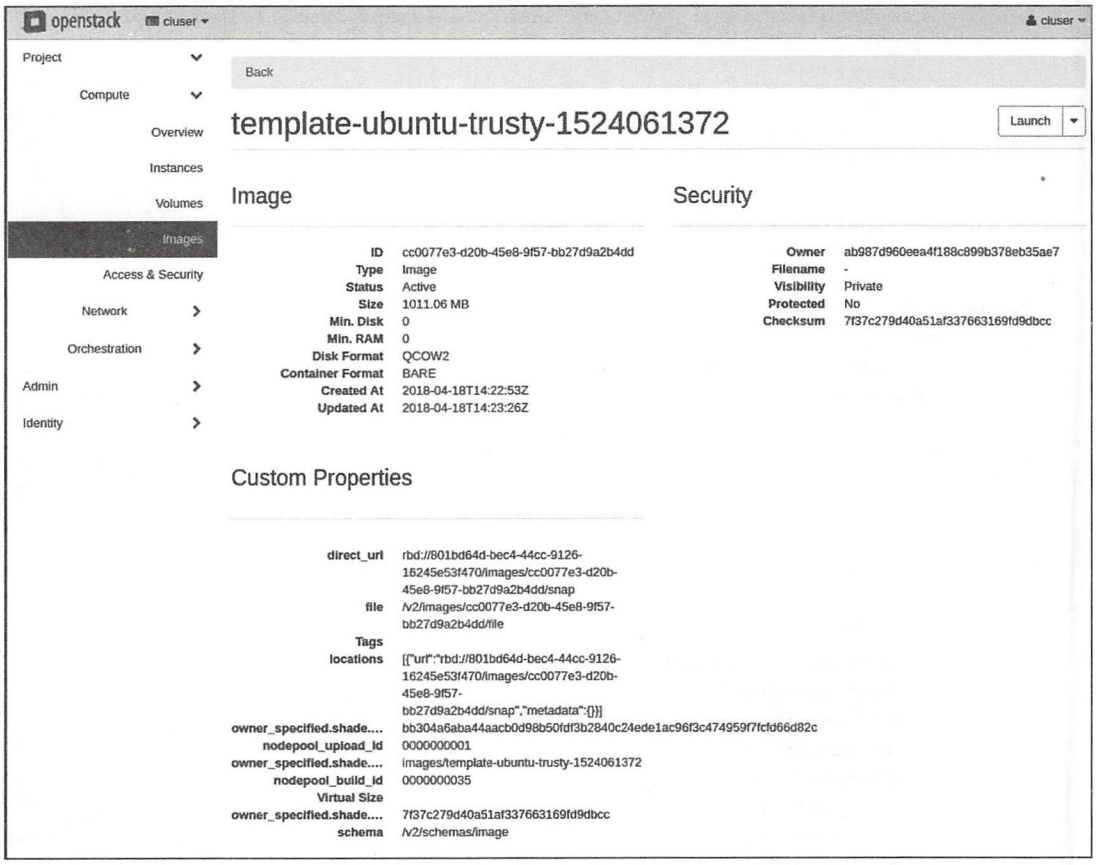


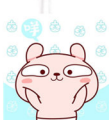
图 10-7　镜像

镜像上传成功之后，Nodepool 使用该镜像孵化节点资源池，登录到云环境上查看，如图 10-8 所示，由虚机名称可以看出这些节点将被注册到 jenkins1，使用的镜像即刚上传的“template-ubuntu-trusty-1524061372”。

节点创建完成后，Nodepool 会自动将其注册到 Jenkins 上，登录 Jenkins 的 Web 页面查看，如图 10-9 所示。

（2）jenkins 目录

主要是配置项目的任务。



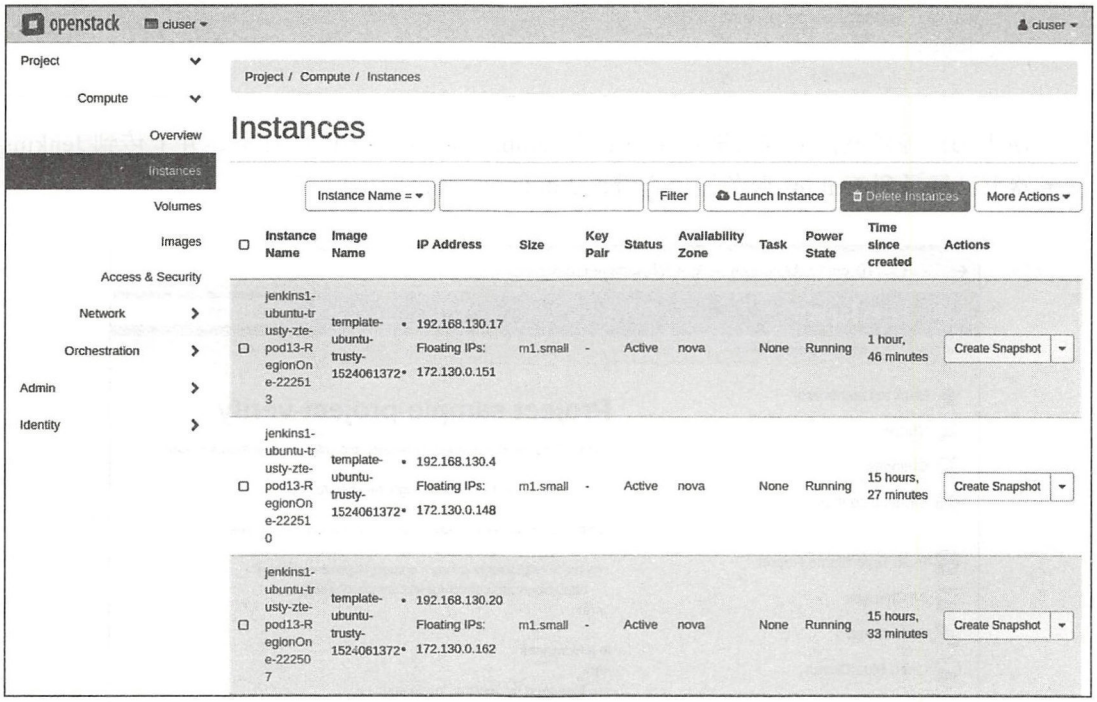


图 10-8　虚机

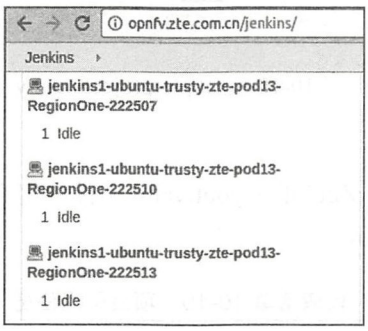


图 10-9　虚机挂载

修改 jobs 目录下的 test.yaml 文件，如代码清单 10-18 所示定义项目的任务。

代码清单 10-18　项目任务定义

```
- job-template:
    name: '{name}-verify'
    node: ubuntu-trusty
    builders:
        - shell: |
            #!/bin/bash -xe
            echo "Job test"
- project:
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
    name: sample-project
    jobs:
        - '{name}-verify'
```

使用 JJB 翻译该任务定义即可生成名为 sample-project-verify 的任务，并上传到 Jenkins 服务器上。登录到 Jenkins 服务器上查看该 Job 信息，如图 10-10 所示。

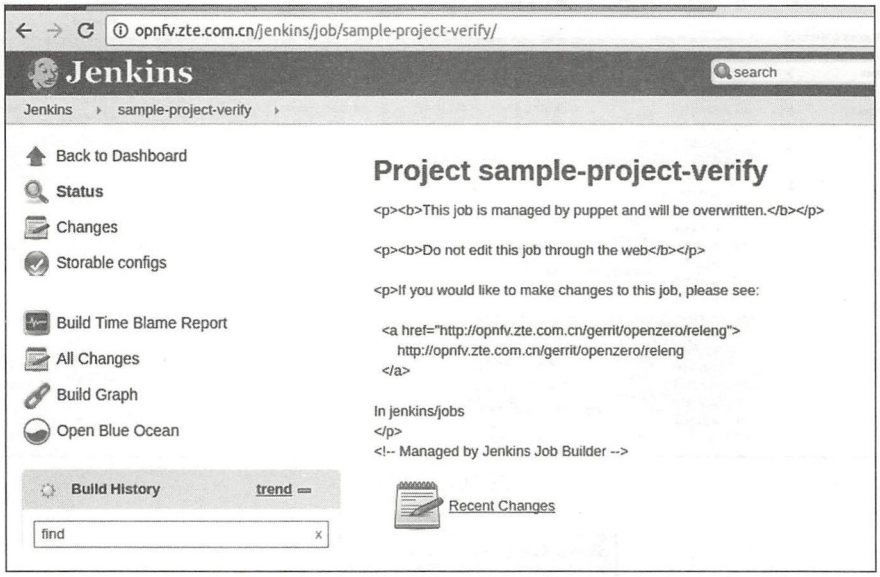


图 10-10　sample-project-verify

（3）zuul 目录

任务定义完成之后即可在 Zuul 的 layout.yaml 文件中定义该项目在什么 pipeline 下运行该任务。如代码清单 10-19 所示。

代码清单 10-19　项目引用任务

```
projects:
    - name: openzero/sample-project
        check:
            - sample-project-verify
        gate:
            - sample-project-verify
```

推送至远程 project-config 仓库，最后再运行如下命令使新配置生效。

```
puppet apply /etc/puppet/manifests/site.pp
```

此时就可以在 Gerrit 的该项目下提交一个变更，查看整个 CI 流程是否工作正常，如图 10-11 所示，可以看到提交了一个新变更即触发了 CI 流程，测试通过后向 Gerrit 返回

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

Verified + 1，整个流程工作正常。

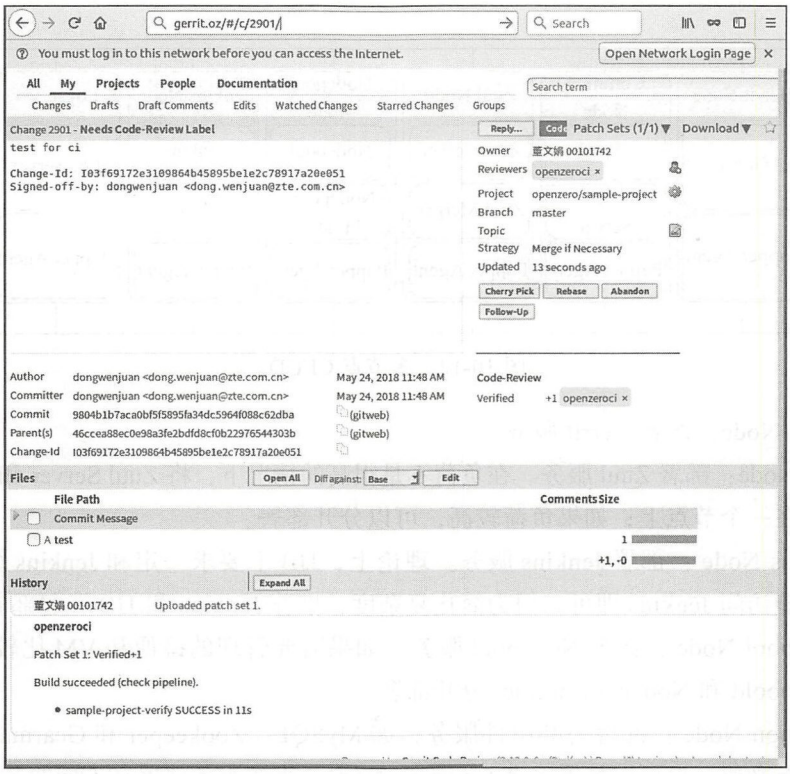


图 10-11　CI 流程验证

10.3　多节点部署

单机部署的搭建和配置相对简单快速，但是不太适用于生产环境，原因如下：

- 组件都运行在同一台服务器上，资源需求较大，特别是 Nodepool 镜像制作会消耗大量的 CPU 和磁盘空间，这势必会影响其他组件的运行。
- 不利于单个组件的升级和维护，比如两个组件具有相同的依赖包，因为升级导致版本不兼容。

为了解决上述问题，考虑按组件服务拆分进行多节点的部署，多节点是 HA 部署的前提。以功能为单位将单节点划分为多个独立的逻辑节点，使用 Puppet Master/Agent 模式，功能节点作为 Puppet Agent，增加 Puppet Master 节点作为配置的中心。Master 和 Agent 都是工作负载节点，为了能部署这些服务并进行有效管理，还需要一个部署服务器运行部署脚本，如本节后面提到的 Ansible 自动化部署脚本就是运行在独立的部署服务器上的，如图 10-12 所示。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

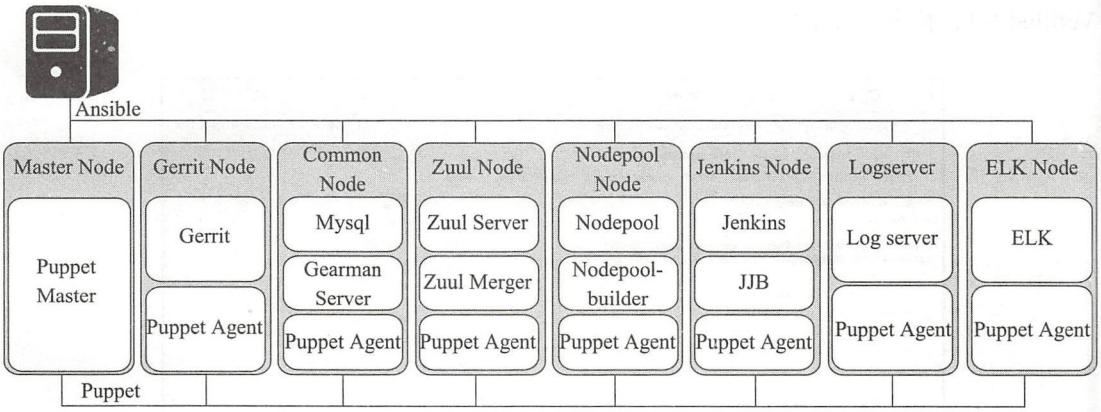


图 10-12 多节点 CI/CD

- Gerrit Node：部署 Gerrit 服务。
- Zuul Node：部署 Zuul 服务。在负荷不是很高的情况下，将 Zuul Server 和 Zuul Merger 部署在一个节点上；如果负荷较高，可以分开部署。
- Jenkins Node：部署 Jenkins 服务。理论上，JJB 不要求一定和 Jenkins 绑定在一起，只要能访问 Jenkins 即可，但为简化复杂度，推荐 Jenkins 和 JJB 一起部署。
- Nodepool Node：部署 Nodepool 服务。如果需要管理的镜像和 VM 比较多，可以将 Nodepoold 和 Nodepool-builder 分开部署。
- Common Node：部署公共组件服务，如 MySQL、Zookeeper 和 Gearman 等。同样，如果有必要，也可以部署多个 Common Node，将各服务部署在不同的节点上。
- LogServer Node：部署日志服务，收集任务构建日志和组件运行日志。
- ELK Node：部署日志分析相关服务，包括 Log pusher、Logstash Indexer、Elasticsearch 和 Kibana 等，部署时可以根据实际负荷将这四个组件部署在一起或者分开。

需要说明的是，这些节点都是按逻辑划分的，具体部署时可以根据实际情况拆分或合并。

10.3.1 IaC

本次部署每个组件之间没有使用 HA，用户如果想实践，只需增加对应的节点即可。准备 8 个 ubuntu 14.04 的虚机，规格：8 核 8G 内存 80G 磁盘，一台作为 Puppet Master，其他作为 Puppet Agent，分别部署 Gerrit、Zuul、Jenkins、Nodepool、公共组件、日志服务器和 ELK。其中日志服务器和 ELK 服务器的磁盘空间可以相对大些。

本次部署遵守基础架构即代码（IaC）的原则，所有的配置和部署脚本都代码化，放在 system-config 库中，代码结构如代码清单 10-20 所示。

代码清单 10-20 system-config 代码结构

```
$ tree
.
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
├── hiera
│   ├── common.yaml
│   └── hiera.yaml
├── install_modules.sh
├── install_puppet.sh
├── manifests
│   └── site.pp
├── modules
│   └── cibook_project
│       ├── files/
│       ├── lib
│       ├── manifests/
│       ├── README
│       └── templates/
├── modules.env
├── README.md
├── setup.cfg
├── setup.py
├── test-requirements.txt
├── tools/
└── tox.ini
```

- hiera 目录：common.yaml 和 hiera.yaml，动态配置数据和 Hiera 配置文件。
- manifests 目录：节点配置文件 site.pp。
- modules 目录：cibook_project 模块，管理整个基础设施，部署组件的入口模块。
- tools 目录：自动化部署脚本。

10.3.2　配置

鉴于单节点部署小节已经详细介绍了各配置文件和参数说明，这里不再重复解释，只介绍下不同之处。

1. 动态配置文件 hiera/common.yaml

多节点的动态配置如代码清单 10-21 所示。

代码清单 10-21　多节点动态配置文件

```
# See parameter documentation inside ../manifests/single_node_ci.pp
# Fields commented out have reasonable default values

#project config repo:
project_config_repo: http://opnfv.zte.com.cn/gerrit/infra/project-config

#vhost_name:
vhost_name_master:      master.cibook.oz
vhost_name_gerrit:      &vhost_name_gerrit gerrit.cibook.oz
vhost_name_zuul:        zuul.cibook.oz
vhost_name_jenkins:     jenkins.cibook.oz
vhost_name_nodepool:    nodepool.cibook.oz
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
vhost_name_mysql:       mysql.cibook.oz
vhost_name_zookeeper:   zookeeper.cibook.oz
vhost_name_gearman:     &vhost_name_gearman gearman.cibook.oz
vhost_name_log_server: &vhost_name_log_server log.cibook.oz
vhost_name_elk:         elk.cibook.oz

#serveradmin:
jenkins_username: cibook
jenkins_password: cibook
jenkins_ssh_private_key: &jenkins_ssh_private_key |
-----BEGIN RSA PRIVATE KEY-----
-----END RSA PRIVATE KEY-----

jenkins_ssh_public_key: &jenkins_ssh_public_key
jjb_git_revision: 1.6.2
jjb_git_url: https://git.openstack.org/openstack-infra/jenkins-job-builder
gearman_server:               *vhost_name_gearman
gerrit_server:                *vhost_name_gerrit
gerrit_user: cibook
gerrit_user_ssh_private_key: *jenkins_ssh_private_key
gerrit_user_ssh_public_key:  *jenkins_ssh_public_key
gerrit_user_http_passwd: cibook
gerrit_ssh_host_key:
git_email: cibook@zte.com.cn
git_name: cibook
log_server:                   *vhost_name_log_server
log_server_public:            *vhost_name_log_server
#smtp_host:
#smtp_default_from:
#smtp_default_to:
zuul_revision: 2.5.2
zuul_git_source_repo: https://git.openstack.org/openstack-infra/zuul

oscc_file_contents: |
# Do Not Edit - Generated & Managed by Puppet
#
# Insert OSCC file contents here as explained in the
# documentation so that nodepool is able to
# authenticate to your cloud(s)
client:
    force_ipv4: true
clouds:
    zte-pod13:
        profile: internap
        floating_ip_source: 'neutron'
        auth:
            username: 'cibook'
            password: 'cibook'
            project_name: 'cibook'
            auth_url: 'http://h13.oz:5000/v2.0'
        regions:
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
            - name: RegionOne
              values:
                networks:
                - name: admin_floating_net
                  routes_externally: true
                - name: net-ci
                  nat_destination: true
                  default_interface: true

#nodepool mysql db config:
mysql_bind_address: '0.0.0.0'
mysql_root_password: root123
mysql_nodepool_db_name: nodepool
mysql_nodepool_password: root123

#gerrit mysql db config:
mysql_gerrit_root_password: UNSET
mysql_gerrit_name: reviewdb
mysql_gerrit_user: gerrit2
mysql_gerrit_password: 12345

nodepool_jenkins_target: jenkins-cibook
jenkins_api_key: 31a5ae595ba5ce364df264bac70e54c0
                                    # jenkins api key get from,!Jenkins
jenkins_credentials_id: 9c72e2be-25c9-4233-a2ac-4dc8664312a4
                                    # jenkins,!credentials_id get from Jenkins
nodepool_revision: 0.4.0
nodepool_git_source_repo: https://git.openstack.org/openstack-infra/nodepool

domain: cibook.oz
#swift_authurl:
#swift_user:
#swift_key:
#swift_tenant_name:
#swift_region_name:
#swift_default_container:
```

对于单机部署，因为只有一台服务器，所以只需要一个 vhost_name 配置；对于多节点部署，需要配置各节点的主机名，新增各节点的主机名参数 vhost_name_{component}，其中 {component} 即各组件名。

日志服务器的配置数据也放入该动态配置文件中。

同时新增 Gerrit 服务部署的配置数据如下：

- mysql_gerrit_root_password：数据库的 root 用户密码，用来登录数据库创建 Gerrit 用户。
- mysql_gerrit_name：Gerrit 的数据库名。
- mysql_gerrit_user 和 mysql_gerrit_password：Gerrit 访问 mysql 的用户名和密码。

其他的配置可参考单机部署相关介绍。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

2. 节点配置文件 manifests/site.pp

所有待部署节点的配置都放入该文件下，节点配置如代码清单 10-22 所示。

代码清单 10-22　多节点节点配置文件

```
node 'gerrit.cibook.oz' {

    $vhost_name_gerrit = hiera('vhost_name_gerrit', $::fqdn)

    package { 'ssl-cert':
        ensure => present,
    }

    exec { 'ensure group ssl-cert exists':
        command => '/usr/sbin/groupadd -f ssl-cert'
    }

    # workaround since pip is not being installed as part of this module
    package { 'python-pip':
        ensure => present,
    }

    class { 'gerrit':
        mysql_host                  => hiera('vhost_name_mysql'),
        mysql_password              => hiera('mysql_gerrit_password'),
        vhost_name                  => $vhost_name_gerrit,
        redirect_to_canonicalweburl => false,
        canonicalweburl             => "https://${vhost_name_gerrit}",
        war                         => 'http://tarballs.openstack.org/ci/g-
errit/gerrit-v2.9.4.5.73392ca.war',
    gerrit_auth_type                => 'DEVELOPMENT_BECOME_ANY_ACCOUNT',
    manage_jeepyb                   => false,
    }
}

node 'jenkins.cibook.oz' {

    $vhost_name_jenkins = hiera('vhost_name_jenkins', $::fqdn)
    class { '::cibook_project::jenkins_node':
        vhost_name                  => $vhost_name_jenkins,
        project_config_repo         => hiera('project_config_repo'),
        serveradmin                 => hiera('serveradmin', "webmaster@${vhost_
name_jenkins}"),
        jenkins_version             => hiera('jenkins_version', 'present'),
        jenkins_vhost_name          => hiera('jenkins_vhost_name', 'jenkins'),
        jenkins_username            => hiera('jenkins_username', 'jenkins'),
        jenkins_password            => hiera('jenkins_password', 'XXX'),
        jenkins_ssh_private_key     => hiera('jenkins_ssh_private_key'),
        jenkins_ssh_public_key      => hiera('jenkins_ssh_public_key'),
        log_server                  => hiera('log_server'),
        jjb_git_revision            => hiera('jjb_git_revision', '1.6.2'),
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
            jjb_git_url                     => hiera('jjb_git_url',https://git.open-
stack.org/openstack-infra/jenkins-job-builder'),
            java_args_override              => hiera('java_args_override', '-Dh-
udson.model.ParametersAction.keepUndefinedParameters=true -Dorg.apache.commons.
jelly.tags.fmt.timeZone=Asia/Shanghai') # lint:ignore:140chars
        }
    }

    node 'log.cibook.oz' {
        class { '::cibook_project::logserver_node':
            domain                      => hiera('domain'),
            jenkins_ssh_key             => hiera('jenkins_ssh_public_key'),
            swift_authurl               => hiera('swift_authurl', ''),
            swift_user                  => hiera('swift_user', ''),
            swift_key                   => hiera('swift_key', ''),
            swift_tenant_name           => hiera('swift_tenant_name', ''),
            swift_region_name           => hiera('swift_region_name', ''),
            swift_default_container     => hiera('swift_default_container', ''),
        }
    }

    node 'zuul.cibook.oz' {
        $vhost_name_zuul = hiera('vhost_name_zuul', $::fqdn)

        class { '::cibook_project::zuul_node':
            vhost_name                  => $vhost_name_zuul,
            project_config_repo         => hiera('project_config_repo'),
            gearman_server              => hiera('gearman_server'),
            gerrit_server               => hiera('gerrit_server'),
            gerrit_user                 => hiera('gerrit_user'),
            gerrit_user_http_passwd     => hiera('gerrit_user_http_passwd'),
            gerrit_user_ssh_public_key  => hiera('gerrit_user_ssh_public_key'),
            gerrit_user_ssh_private_key => hiera('gerrit_user_ssh_private_key'),
            gerrit_ssh_host_key         => hiera('gerrit_ssh_host_key'),
            git_email                   => hiera('git_email'),
            git_name                    => hiera('git_name'),
            log_server                  => hiera('log_server'),
            log_server_public           => hiera('log_server_public'),
            smtp_host                   => hiera('smtp_host', 'localhost'),
            smtp_default_from           => hiera('smtp_default_from', "zuul@-
${vhost_name_zuul}"),
            smtp_default_to             => hiera('smtp_default_to', "zuul.reports-
@${vhost_name_zuul}"),
            zuul_revision               => hiera('zuul_revision', 'master'),
            zuul_git_source_repo        => hiera('zuul_git_source_repo'),
        }
    }

    node 'nodepool.cibook.oz' {
        # If the fqdn is not resolvable, use its ip address
        $vhost_name_nodepool = hiera('vhost_name_nodepool', $::fqdn)
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
        class { '::cibook_project::nodepool_node':
            vhost_name                      => $vhost_name_nodepool,
            project_config_repo             => hiera('project_config_repo'),
            jenkins_username                => hiera('jenkins_username', 'jenkins'),
            jenkins_ssh_public_key          => hiera('jenkins_ssh_public_key'),
            jenkins_ssh_private_key         => hiera('jenkins_ssh_private_key'),
            oscc_file_contents              => hiera('oscc_file_contents', ''),
            mysql_host                      => hiera('vhost_name_mysql'),
            mysql_root_password             => hiera('mysql_root_password'),
            mysql_db_name                   => hiera('mysql_nodepool_db_name', 'node-
pool'),
            mysql_nodepool_password         => hiera('mysql_nodepool_password'),
            nodepool_jenkins_target         => hiera('nodepool_jenkins_target', 'jenkins-
cibook'),
            jenkins_api_key                 => hiera('jenkins_api_key', 'XXX'),
            jenkins_credentials_id          => hiera('jenkins_credentials_id', 'XXX'),
            jenkins_url                     => hiera('jenkins_url', 'http://jenkins-
.cibook.oz:8080'),
            nodepool_revision               => hiera('nodepool_revision', 'master'),
            nodepool_git_source_repo        => hiera('nodepool_git_source_repo',-
http://opnfv.zte.com.cn/gerrit/openstack/nodepool'),
        }
    }

    node 'elk.cibook.oz' {

        $elasticsearch_nodes = [ hiera('vhost_name_elk') ]

        class { 'cibook_project::logstash':
            discover_nodes                  => [ "hiera('vhost_name_elk'):9200" ]
        }

        class { 'cibook_project::logstash_worker':
            discover_node                   => hiera('vhost_name_elk'),
            enable_mqtt                     => false,
            mqtt_hostname                   => hiera('vhost_name_elk'),
            mqtt_password                   => '',
            mqtt_ca_cert_contents           => '',
        }

        class { 'cibook_project::elasticsearch_node':
            discover_nodes => $elasticsearch_nodes
        }
    }

    node 'gearman.cibook.oz' {
         class { 'gearman': }
    }

    node 'zookeeper.cibook.oz' {
        class { '::cibook_project::common_nodepool_db':
```

```
        vhost_name                      => hiera('vhost_name_zookeeper', $::fqdn),
        mysql_bind_address              => hiera('mysql_bind_address'),
        mysql_root_password             => hiera('mysql_root_password'),
        mysql_nodepool_password         => hiera('mysql_nodepool_password'),
    }

    class { 'zookeeper':
        install_java=> true,
        java_package => 'openjdk-7-jre-headless',
    }
}
```

各节点调用 cibook_project 模块中的类完成部署。

3. 部署模块 modules/cibook_project

openstackci 模块是单机部署的入口模块，在 single_node_ci 类中部署了 Jenkins、Zuul 和 Nodepool，对于多节点部署只需要把它们按组件拆开成独立的类，新增 cibook_project 模块来完成这些类的管理，其他 Gerrit、ELK 和日志服务器等相关资源管理也放在此模块下。

Puppet 模块的调用关系如图 10-13 所示。

配置完成之后，下面进行自动化部署。

10.3.3　自动化部署

在部署服务器上使用 Ansible 自动化部署工具编排待部署的节点：

- 安装和配置 Puppet，完成 Agent 向 Master 的自动认证。
- 驱动 Puppet 自动化安装部署各节点上的 CI 应用服务。

部署脚本位于 system-config/tools 目录下，如代码清单 10-23 所示。

代码清单 10-23　ansible 部署脚本代码结构

```
$ tree
.
├── ansible.cfg
├── inventory
├── puppetagents.yml
├── puppetmaster.yml
└── roles
    ├── puppetagent
    │   ├── defaults
    │   │   └── main.yml
    │   ├── tasks
    │   │   └── main.yml
    │   └── templates
    │   └── puppet.conf.j2
    ├── puppetmaster
    │   ├── defaults
    │   │   └── main.yml
    │   ├── handlers
```

```
|   |   └── main.yml
|   ├── tasks
|   |   └── main.yml
|   └── templates
|   ├── autosign.conf.j2
|   └── puppet.conf.j2
└── puppet-node
└── tasks
    └── main.yml
```

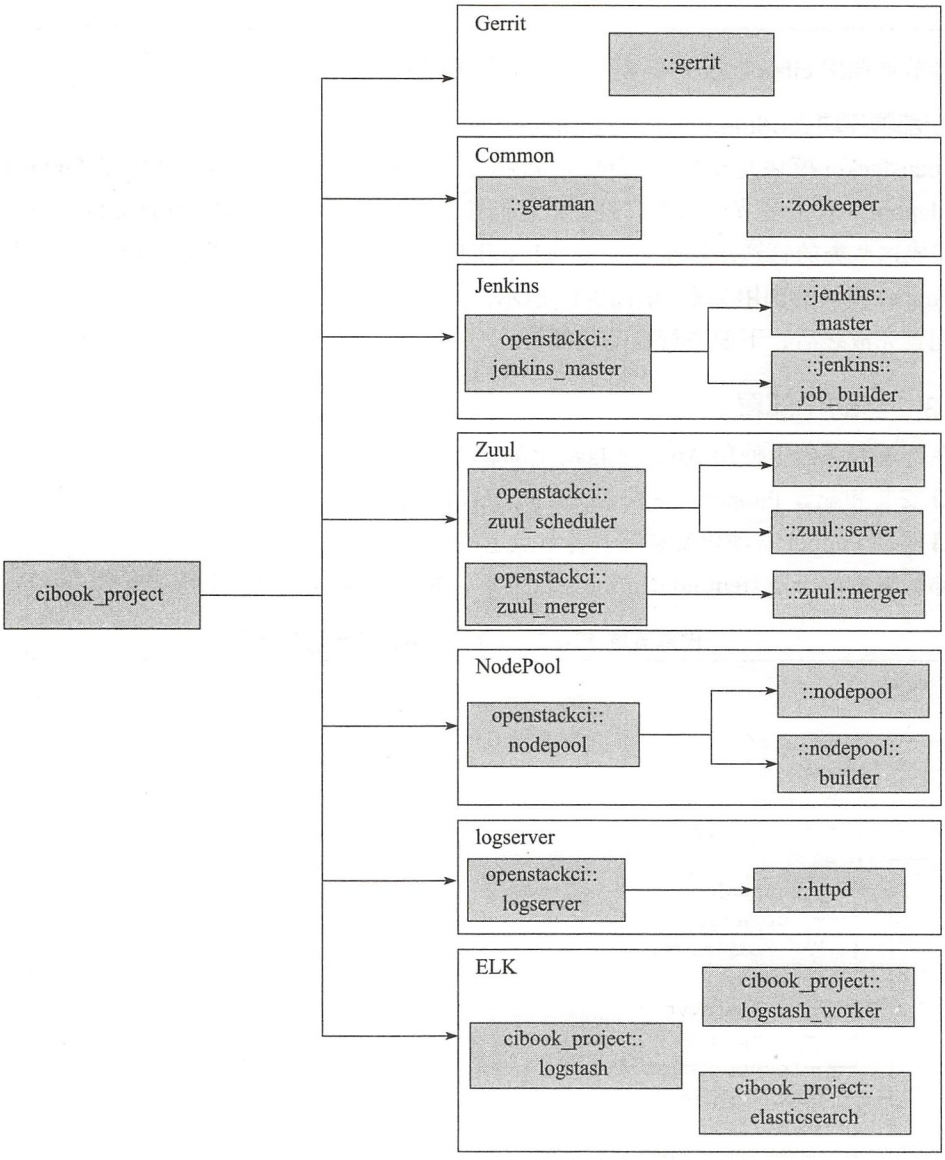


图 10-13　多节点部署模块调用关系

各部署脚本不做详细说明。

首先在部署服务器上安装 Ansible：

```
sudo apt-add-repository -y ppa:ansible/ansible
sudo apt-get update -y
sudo apt-get install -y ansible
```

然后复制 system-config 库到部署服务器上，进入 tools 目录下，执行如下命令即可运行部署：

```
ansible-playbook puppetmaster.yml
ansible-playbook puppetagents.yml
```

Ansible 内部部署流程如图 10-14 所示。

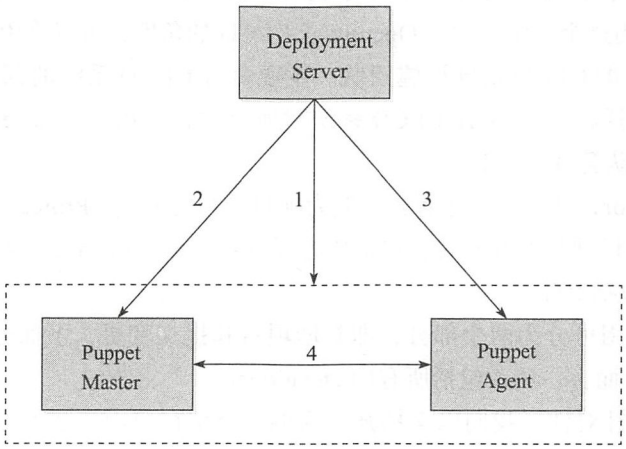


图 10-14　多节点自动化部署流程

1. 所有待部署节点

1）在所有待部署节点上安装 NTP 服务，防止由于时钟不同步导致 Puppet Agent 和 Puppet Master 同步失败。

2）修改宿主机名称。

2. Master 节点

1）安装 puppetmaster 服务。

2）设置自动签名认证。

3）修改 puppet.conf 文件，重启 puppetmaster 服务使配置生效。

3. Agent 节点

1）安装 puppet 服务。

2）修改 puppet.conf 文件，指定 Master 服务器地址以及本机向 Master 认证的名称。

3）使能 puppet 服务。

4. Puppet Agent 和 Puppet Master 交互

1）Agent 向 Master 发送认证请求，Master 根据 Agent 携带的主机名决定是否认证签名。

2）认证签名通过，Agent 携带主机名和 facts 向 Master 请求 catalog。

3）Master 分析客户端的主机名，找到该主机的配置代码，编译 catalog 并发回客户端。

4）Agent 执行代码完成各节点组件部署，并把执行情况反馈给 Master。

最后由用户根据执行结果查看部署情况，这里不再赘述。

10.4 使用 CI/CD

到目前为止，我们只是建立了一个具有 CI/CD 基础设施的框架，里面没有任何内容，下面我们介绍如何使用这个系统。站在 OpenStack CI/CD 的角度，可以将用户分成以下几类：

- Infra team，即 CI/CD 基础设施团队。主要负责 CI/CD 系统的部署和维护（如扩容、数据备份和迁移等），保证 CI/CD 系统对所有项目可用。Infra team 日常工作也包括协助项目团队管理 CI/CD。
- Project creator，即项目创建者，一般为项目最初的 PTL（Project Team Leader）；
- Developer，即项目的开发者，包括核心开发者（core member）和普通开发者（developer），也包括 PTL。

对 CI/CD 的使用可分为两个部分，即新增项目和提交变更，下面分别加以介绍，前者针对 Project creator 而言，后者包括所有的 Developer。

为使描述更有针对性，我们以文档开发为例，介绍在前面搭建的 CI/CD 系统中如何创建项目、如何提交变更；为了便于评审和版本控制，我们使用 rst 格式编写文档，每次变更都会触发 CI/CD 对文档进行语法检查、自动文档构建，并自动将文档以 html 和 pdf 格式发布到对应的服务器上。整个示意图如图 10-15 所示。

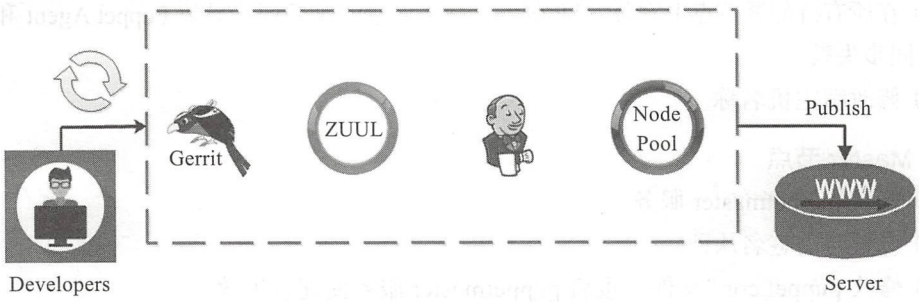


图 10-15　通过 CI/CD 进行文档开发示意图

10.4.1 新增项目

通常增加项目需要考虑以下几个步骤：

1）Gerrit 中增加项目并作相关配置。

2）Zuul 中配置 pipeline。

3）Jenkins 中增加 Jobs。

4）Nodepool 中增加标签类型或镜像。

如果使用的 Gerrit 中已有项目，也需要增加相关权限配置；Nodepool 中的配置项可以根据实际需求调整。

项目创建一般由 Project creator 发起，由 Infra team 成员配合执行（如检查新加入的配置对其他项目是否影响）。

注解　如非特殊说明，下面所有的配置都是修改 project_config 项目下的文件。地址为 https://github.com/openzero-team/project_config.git

---

1. Gerrit 中新增项目

待增加的项目可以继承已经存在的项目，或者是全新的项目，不管哪种情况，要使用 CI/CD 系统，第一步需要将项目增加到 Gerrit 中，大致包括以下几个步骤：

1）添加项目。

2）增加项目用户组。

3）分配用户组权限。

4）创建 / 删除项目分支。

Gerrit 的配置修改主要涉及 gerrit/projects.yaml 文件和 gerrit/acls/ 文件夹下的文件。前者是 Gerrit 所有项目及配置列表，后者是各对应项目的权限配置；对于配置文件中不支持的操作（如创建项目分支或标签），可以使用 Gerrit CLI 或在 Web UI 上配置。

（1）添加项目

在 Gerrit 中添加项目主要包括两种情况：

1）新增项目。修改 gerrit/projects.yaml，在该文件中增加项目定义，如下所示：

```
- project: <project_namespace>/<projectname>
    description: <some descriptions>
```

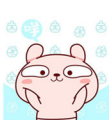
2）创建既有项目。如果项目库已经存在（如待加入的项目是基于某个上游项目），有两种方法：

❑ 使用 upstream 将项目导入，如下所示：

```
- project: <project_namespace>/<projectname>
    description: <some descriptions>
    upstream: <upstream url>
```

❑ 首先在 Gerrit 上创建空项目，然后从上游将对应的项目复制到本地，再 push 到 Gerrit 上。

如果需要保留所有历史，推荐使用 upstream 字段导入的方式；而使用 push 的方式可以



只保留最近一部分历史，从而减少空间占用。用户可根据实际情况选择。

创建空项目也可以通过 Gerrit 提供的 Web UI 直接操作或者通过 CLI，具体参见 3.2 节中的描述。

对于本文档项目添加如下：

```
- project: openzero/openzero
    description: Configuration files for CI/CD book
```

（2）增加项目用户组

对于一个项目，一般需要创建如表 10-1 所示的用户组。

表 10-1 项目用户组

序号	用户组	职责
1	<projectname>-core	核心成员组，拥有 + 2/– 2 权限
2	<projectname>-manager 或者 <projectname>-release	版本管理组，拥有打标签的权限

以上只是 OpenStack 社区推荐的默认组，可以根据实际项目管理的需要增加其他的组，如 <projectname>-owner 组描述项目的所有者，其中只包含 PTL 或少数几个核心成员，对项目拥有最高的权限。

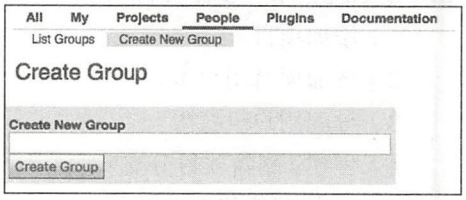


图 10-16 通过 Web 创建用户组

增加用户组可以使用 Web UI 方式：以管理员的身份登录 Gerrit Dashboard，通过 People → Create New Group 创建，如图 10-16 所示。

也可以通过 CLI 的方式创建：

```
ssh -p 29418 <administrator>@<gerritserver> gerrit create-group <group-name>
```

其中 <administrator> 是 Gerrit 管理员名称，<gerritserver> 是 Gerrit 服务器的地址，<group-name> 是需要创建的组名，即 <projectname>-core 或者 <projectname>-manager 等。创建完用户组后，可以为组添加用户，可以通过 Web 方式进行，如图 10-17 所示。

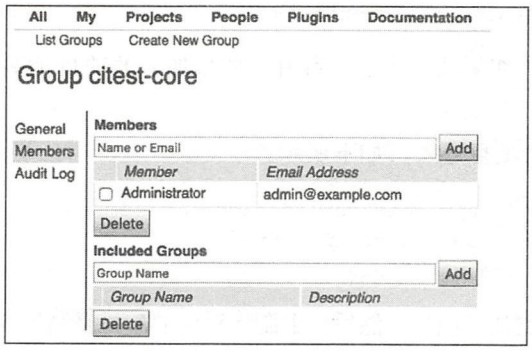


图 10-17 为组添加用户

为用户组添加成员时，除了直接添加用户外，还可以引用其他的组。

注解　添加到组中的用户必须是已经在 Gerrit 中注册的用户。

为 openzero 添加的用户组如图 10-18 所示。

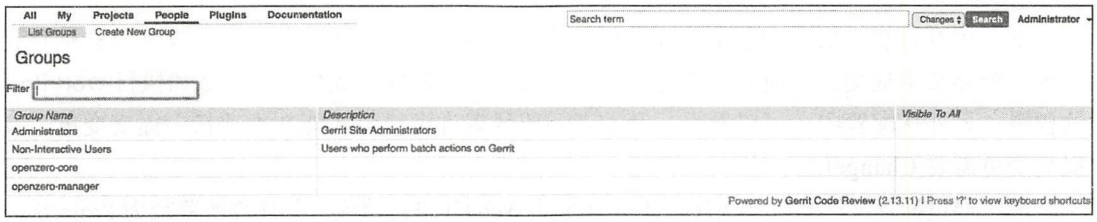


图 10-18　opnezero group

openzero 项目添加了 openzero-core 和 openzero-manager 两个用户组。

为 openzero 添加的用户如图 10-19 所示。

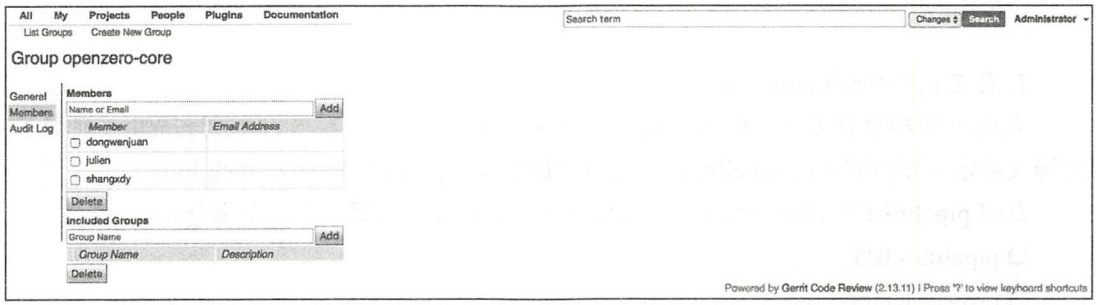


图 10-19　opnezero members

（3）添加项目权限

Gerrit 权限都是基于用户组的，一旦为用户组赋予了某种权限，该组中所有的用户都将具有对应的权限；关于项目权限可以参见 3.2 节中的描述。对于指定的项目权限配置，需在项目下增加文件 gerrit/acls/openzero/openzero.config，如代码清单 10-24 所示：

代码清单 10-24　openzero 项目权限配置

```
[access "refs/heads/*"]
abandon = group openzero-core
create = group openzero-manager
label-Code-Review = -1..+1 group Registered Users
label-Code-Review = -2..+2 group openzero-core
label-Workflow = -1..+1 group openzero-core

[access "refs/tags/*"]
```

```
pushSignedTag = group openzero-manager

[receive]
requireChangeId = true
requireContributorAgreement = true

[submit]
mergeContent = true
```

配置中允许核心成员和版本管理员创建分支，但是只允许版本管理员创建标签；核心成员可以废弃某个提交，普通开发者有 -1/+1 的权限，但是核心成员有 -2/+2 和执行 workflow 的权限；另外从配置中还可以看出，开发者需要签署许可协议才能在本项目中提交变更，且提交变更需要 ChangeId。

除了通过配置文件修改权限外，还可以通过 Web UI 进行修改，具体修改路径为 Projects → Access。

（4）增加 / 删除项目分支

项目默认当前的分支为 master 分支，可能会根据情况创建其他分支，如 dev。不能通过配置文件为项目增加或删除分支，需要通过 CLI 或者 Gerrit Web UI 进行操作，具体参见 3.2 节中的描述。

2. 在 Zuul 中配置 pipeline

完成了新项目在 Gerrit 中的配置，接下来需要在 Zuul 中为项目定义 pipeline 规则了：选择或新定义项目需要的 pipeline，并说明项目在该 pipeline 需要执行哪些 Job。

Zuul pipeline 的配置位于 zuul/layout.yaml 中，该文件包括以下几个部分：

- pipelines 配置。
- project-templates 配置。
- jobs 配置。
- projects 配置。

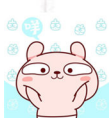
其中前三个配置都是为第四个 projects 配置服务的，且必须对 pipelines 和 projects 进行配置，project-templates 和 jobs 两部分可以根据实际情况选择配置。

（1）配置 pipelines

该过程主要是为项目选择合适的工作流，工作流由一系列 pipeline 组成，OpenStack CI/CD 提供了一些通用的 pipeline[⊖]，可以将这些定义复制过来直接使用，本节不对这些 pipeline 定义作详细解释，仅对各 pipeline 的使用做以下说明：

- check：新的变更提交后，会进入此 pipeline，运行对应项目在此 pipeline 下的 Job 后，根据测试结果，自动给出 +1/-1 的评价；该 pipeline 一般用来进行代码静态检查、单元测试等。

⊖ https://github.com/openstack-infra/project-config/blob/before-jenkins-config-removed/zuul/layout.yaml。



非卖品，仅供非商业用途或交流学习使用

- gate：变更被核心成员批准后会进入此 pipeline，如果项目在该 pipeline 下的 Job 都运行成功，变更会被合入目标分支；该 pipeline 一般用来进行集成测试。
- post：合入目标分支后的变更会进入该 pipeline，该 pipeline 一般用来发布临时文档或软件，或者用来将变更部署到非正式环境（如测试环境）中。
- pre-release：项目打上预发布标签（标签名称以“rc”结尾）后，会进入该 pipeline；一般使用该 pipeline 来处理项目版本预发布，如推送预商用版本到制品库，或者将版本部署到预商用环境。
- release：项目打上发布标签（标签名称不以“rc”结尾）后，会进入该 pipeline；一般使用该 pipeline 来处理版本正式发布，如推送正式版本到制品库，或者在正式环境中部署项目。
- silent：该 pipeline 一般用于测试 Job，只有测试无误后才允许进入 experimental pipeline；变更需要手动触发才能进入该 pipeline。
- experimental：该 pipeline 一般用于测试 Job，测试通过后才允许进入 check pipeline；变更需要手动触发才能进入该 pipeline。
- periodic：周期性触发的 pipeline，对于需要周期性运行 Job 的项目，可以使用该 pipeline；该 pipeline 和具体变更无关，Job 是由设定的时间周期性触发的，如每天编译版本或每隔一段时间运行一次测试任务。

通过这些 pipeline 可以为项目组织工作流，典型的工作流如下：

check → gate → post → pre-release → release。

注解　为项目新增 Job 时，如果不确定 Job 是否可以稳定运行，社区推荐的做法是：

- 默认使用 silent pipeline，通过增加 recheck silent 或 reverify silent 评论手动触发 Job 运行；
- 如果 silent 下 Job 运行都成功，将 Job 移动到 experimental pipeline 下，通过增加 check experimental 评论手动触发 Job 运行；
- 如果 experimental 下 Job 运行也成功，则意味着可以将 Job 移动到 check pipeline，且 Job 配置为 non-voting 模式，然后通过变更自动触发 Job，并分析其结果；
- 只有 check pipeline 运行成功后，再增加 gate pipeline，并将 Job 配置为 voting 模式。

如果上面提供的 pipeline 不能满足项目的要求，可以根据需要自定义 pipeline。我们假设下面的一个场景：变更合入到目标分支后，会触发新的部署，但有时可能部署会失败，且失败不是因为变更导致的（如偶尔网络的问题），按照一般的 pipeline 流程，为了定位这类部署失败的真正原因，需要按照 check → gate → post 工作流程依次触发所有 Job 的执行，这个过程可能会比较久，有时也没有必要，为了解决这类问题，我们在 post 后定义一个新的 pipeline，支持手动触发软件部署，即整个 pipeline 流程变为 check → gate → post|redeploy。

非卖品，仅供非商业用途或交流学习使用

pipeline 的前后变化如图 10-20 所示。

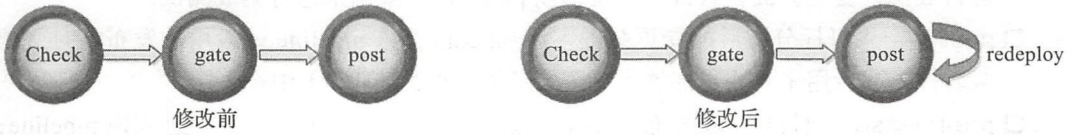


图 10-20　通过 pipeline 改变工作流

为 openzero 新加入的 pipeline 定义如代码清单 10-25 所示：

代码清单 10-25　定义新的 pipeline-redeploy

```
    - name: redeploy
        description: >
            Have a change to redeploy the patchset when deployment procedure runs wrong.
        failure-message: Deployment failed (redeploy pipeline).
        manager: IndependentPipelineManager
        ignore-dependencies: true
        source: gerrit
        precedence: low
        require:
            approval:
                - verified: [1, 2]
            open: false
        trigger:
        gerrit:
            - event: comment-added
              comment: (?i)^(Patch Set [0-9]+:)?( [\w\\+-]*)*(\n\n)?\s*(redeploy|
forecedeploy)
```

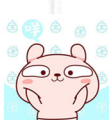
这样，如果一个项目使用了 redeploy pipeline，在新的变更合入后，对于出现自动部署失败的情况，只需要在变更中增加 redeploy 或者 forcedeploy 的评论，就会重新触发部署流程。

（2）配置 project-templates

项目模板的定义位于 project-templates: 中。配置 project-templates 包括直接在新增项目中引用已有的模板定义，也包括定义新的模板。代码清单 10-26 是为 openzero 定义项目模板。

代码清单 10-26　openzero 项目模板

```
    project-templates:
        - name: book-rst-jobs
            check:
                - '{name}-docbuild-rst'
            gate:
                - '{name}-docbuild-rst'
            post:
                - '{name}-docbuild-rst-release'
```



非卖品，仅供非商业用途或交流学习使用

新定义的模板名称为 book-rst-jobs，使用了 check、gate 和 post 三个 pipeline，其中前两个 pipeline 完成文档的检查和编译，最后一个完成文档的发布。

（3）配置 projects

有了 pipeline，定义好了 project-templates，就可以在项目中定义需要运行哪些 Job 了。project 的配置在 projects: 中。有两种方式定义项目中的 Job，一是通过 template 引用 project-templates 中的定义，另一种是直接在项目中定义指定 pipeline 下的 Job 列表。如代码清单 10-27 所示。

代码清单 10-27　为 openzero 定义 projects

```
- name: openzero/openzero
    template:
        - name: book-rst-jobs
    redeploy:
         - '{name}-docbuild-rst-release'
```

代码清单 10-27 中通过模板引用了 project-templates 中的模板定义即 book-rst-jobs，并且为 redeploy pipeline 增加了 Job {name}-docbuild-rst-release，这个定义和代码清单 10-28 中的定义是一个效果。

代码清单 10-28　展开后的 projects

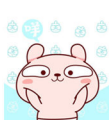
```
- name: openzero/openzero
    check:
        - '{name}-docbuild-rst'
    gate:
        - '{name}-docbuild-rst'
    post:
        - '{name}-docbuild-rst-release'
    redeploy:
        - '{name}-docbuild-rst-release'
```

注解　Zuul 中定义的只是 Job 触发条件，而不定义 Job 的具体执行行为，Job 的具体行为是在 Jenkins 中定义的。

（4）配置 Jobs

Jobs 的配置位于 jobs 中，主要对用户运行 Job 的属性做一些调整，如 Job 的运行结果是否参与投票，Job 只关心项目下某些文件的变更，或者指定 Job 只运行在特定的分支下等。如下代码定义了运行名称为 execute-jenkins-view-apply-in-cimaster 的 Job 时，不需要根据运行结果进行投票：

```
- name: execute-jenkins-view-apply-in-cimaster
    voting: false
```



非卖品，仅供非商业用途或交流学习使用

注解 不投票的配置建议只在拥有 check pipeline 的 Job 中增加，不应该为 gate 下的 Job 增加该属性，因为在 gate 中增加这样的 Job 是一种资源的浪费。

代码清单 10-29 是为 openzero 定义的 Job 过滤规则。如果 openzero 项目中 docs 目录或 doc 目录下的文件变化，opezero-docbuild-rst 和 opezero-docbuild-rst-release 两个 Job 会触发运行，但会忽略 docs/com/ 目录下的文件的变化，并且项目根目录下 rst 后缀的文件发生变化也会触发这两个 Job 运行。

代码清单 10-29 openzero Job 过滤规则

```
jobs:
    - name: ^.*-docbuild-rst*
        files:
            - ^docs/.*$
            - ^doc/.*$
            - .*\.rst
        skip-if:
            - all-files-match-any:
                - ^docs/com/.*$
```

3. 在 Jenkins 中添加任务

项目真正需要执行的 Job 定义在 Jenkins 中，这些 Job 的定义位于 jenkins/jobs/ 目录中，该目录包含 4 类文件：

- defaults.yaml：Jenkins Jobs 默认参数定义文件。
- macros.yaml：Jenkins Jobs 宏定义文件。
- <project_name>.yaml：特定项目的 Job 配置文件。
- projects.yaml：项目模板的实例化文件。

对于特定项目，除了 <project_name>.yaml 这个文件外，其他 3 类文件可以根据实际需要选择配置。对于 openzero 项目，需要增加 openzero.yaml 文件定义。

（1）Defaults

该配置位于 defaults.yaml 文件中，以 -defaults: 行开始。如代码清单 10-30 是一个定义示例。

代码清单 10-30 Jenkins default 定义

```
- defaults:
    name: global
    project-type: freestyle
    concurrent: true

    wrappers:
        - timeout:
            timeout: 30
```

非卖品，仅供非商业用途或交流学习使用


```
            fail: true
        - timestamps

    logrotate:
        daysToKeep: 1
        numToKeep: -1
        artifactDaysToKeep: -1
        artifactNumToKeep: -1
```

示例中 global 表明该默认配置全局有效，项目类型是 freestyle，允许 Jobs 并行执行，Job 执行的超时时间是 30 分钟，日志上报时需要打时间戳，定义的最后还指定了日志的默认保存策略。可以根据需要增加其他参数定义。

（2）Macros

该配置位于 macros.yaml 文件中，宏为 Job 定义的复用提供了很大的方便，在使用时直接使用宏名称即可。前文配置 pipeline 时为 openzero 项目定义了两个 Job，每个 Job 都会复制待测项目、上传日志等，我们把这些公共的部分定义为宏，如代码清单 10-31 所示的是为 openzero 项目增加的宏定义。

代码清单 10-31　openzero Jenkins Macros 定义

```
- builder:
    name: zuul-git-prep
    builders:
        - shell: |
            #!/bin/bash -xe
            CLONEMAP=`mktemp`
            function cleanup {
                    mkdir -p $WORKSPACE
                    rm -f $CLONEMAP
            }
            trap cleanup EXIT
            cat > $CLONEMAP << EOF
            clonemap:
                - name: $ZUUL_PROJECT
                    dest: .
            EOF
            /usr/zuul-env/bin/zuul-cloner -m $CLONEMAP --cache-dir /opt/git \
                ssh://openzeroci@gerrit.oz:29418 $ZUUL_PROJECT

- publisher:
    name: console-log
    publishers:
        - scp:
            site: 'LogServer'
            files:
                - target: 'logs/$LOG_PATH'
                    copy-console: true
                    copy-after-failure: true
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

宏 zuul-git-prep 完成项目在 Slave 上的复制操作，宏 git-clone-opnfv-scripts 会下载需要的运行脚本。

（3）Job

对于 Job 定义，除了直接定义 Job 外，还可以将 Job 定义为模板，以便多个 Job 复用；Job 模板的定义一般也是放在项目特有的定义文件 openzero.yaml 中。通过模板，可以批量定义 Job，和 Zuul 中的项目模板一样，定义 Job 模板一般也会携带变量，供具体实例化时使用。代码清单 10-32 是 Job 模板定义示例。

代码清单 10-32 openzero Jenkins Job 模板定义

```
- job-template:
    name: '{name}-docbuild-rst'
    node: ubuntu-trusty
    doc_dir: "docs"
    builders:
        - zuul-git-prep
        - shell: |
            #!/bin/bash -xe
            sudo apt-get update
            sudo apt-get install -y graphviz
            export DOCS_DIR='{doc_dir}'
            /tmp/_releng/docs/scripts/docs-build.sh
    publishers:
        - console-log

- job-template:
    name: '{name}-docbuild-rst-release'
    node: ubuntu-trusty
    doc_dir: "docs"
    builders:
        - zuul-git-prep
        - shell: |
            #!/bin/bash -xe
            export DOCS_DIR='{doc_dir}'
            /tmp/_releng/docs/scripts/docs-build.sh

            # remain issue:
            # java can not upload docs_output to octopus.oz
            scp -r docs_output/* deploy@octopus.oz:/home/deploy/docs/{name}
    publishers:
        - console-log
```

openzero 定义了两个项目模板，即‘{name}-docbuild-rst’和‘{name}-docbuild-rst-release，’模板中都引用两个宏定义 zuul-git-prep 和 git-clone-opnfv-scripts，并增加文档发布和日志上报两个宏。

（4）project

如果定义了 Job 模板，要将这些模板实例化为具体的 Job，就必须定义 project，project

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

的定义位于文件 projects.yaml 中。代码清单 10-33 展示了代码清单 10-32 具体实例化的过程。

代码清单 10-33　openzero Jenkins project 定义

```
- project:
    name: openzero
    project: '{name}'
    reveal_path: 'reveal'
    jobs:
        - '{name}-docbuild-rst'
        - '{name}-docbuild-rst-release'
```

4. 修改 Nodepool 配置

对于新增项目，一般不涉及 Nodepool 的配置，但是为项目定义的 Job 如果需要运行在特殊的 Slave 上，或者项目 Job 运行需要特殊的镜像支撑，就需要重新制作镜像、从云中创建 VM 并打上新的标签类型。这种情况下，会涉及 nodepool.yaml 中 label、provider 和 diskimage 三个配置项的变更，该配置位于 nodepool/nodepool.yaml 中，在前文 10.2.2 节中已经有相关解释，此处不再赘述，Nodepool 的具体配置及其含义请参见第 6 章。

10.4.2　提交变更

对于普通开发人员来说，一般不会增加项目，更多的是为既有项目做贡献，为了能提交变更到项目中，需要做一些准备工作，包括创建 Gerrit 账户、安装配置 git-review 工具等，下面分别加以说明。

1. 准备工作

（1）创建 Gerrit 用户

要向系统提交变更，需要创建一个 Gerrit 账户，为简化处理，Gerrit 没有对接认证系统，所以普通用户还不能自己创建，只能由具有管理权限的用户代为创建。可以通过 Gerrit Web UI 创建账户，如图 10-21 所示。

创建时需要提供用户名、email 和 SSH 公钥等。

除了 Web UI 外，还可以通过 CLI 的方式创建账户，如代码清单 10-34 所示。

代码清单 10-34　通过 CLI 创建 Gerrit 用户

```
cat <your ssh pub key file> | ssh -p 29418 <admin name>@<gerrit-serer> gerrit \
    create-account <acount name> --email <your email> --full-name <your name> \
    --http-password <default password> --ssh-key -
```

创建完账号后，按照如下方式配置 git：

```
git config --global user.name <acount name>
git config --global user.email <your email>
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

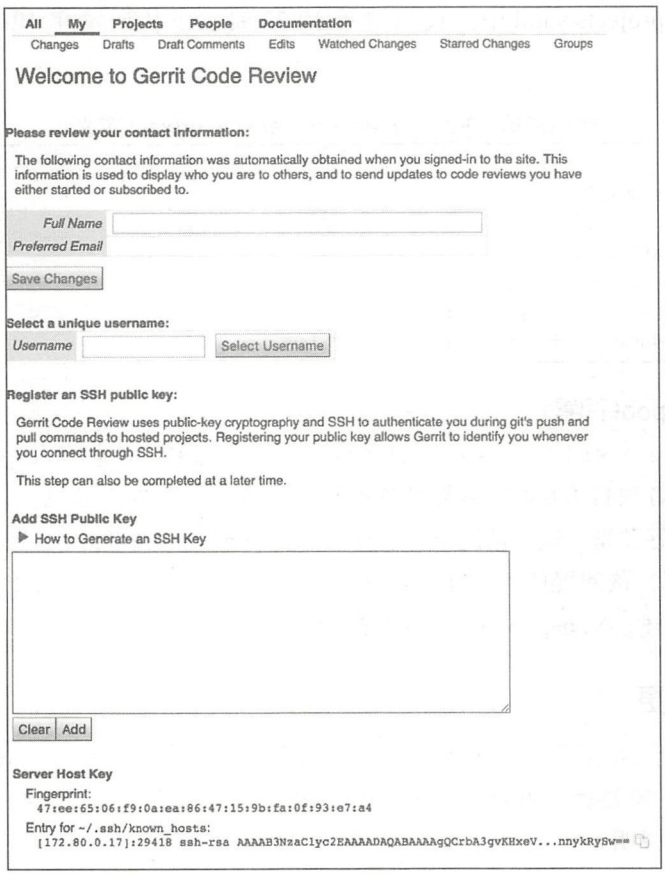


图 10-21　通过 Web 创建 Gerrit 用户

（2）安装 git-review

开发环境使用 git-review 作为代码评审提交工具，通过下面方式安装：

```
apt-get install git-review
```

2. 为新项目做贡献

为新项目做贡献，首先需要复制代码，如下所示：

```
git clone ssh://shangxdy@gerrit.cibook.oz:29418/openzero/openzero
```

接着需要为项目配置 Gerrit（此时也可以不配置，在第一次提交变更时 Gerrit 会提示增加配置）：

```
cd openzero
git config --global gitreview.username <acount name>
git remote add gerrit https://<acount name>@<project_repository_url>
git review -s
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

最后一行命令会检查用户是否可以通过 SSH 公钥登录 Gerrit，如果成功没有任何返回，否则会返回错误信息。

项目配置完毕之后，在本地创建并检出分支，然后可以修改代码了：

```
git checkout -b <topic-branch>
```

修改完毕并经过本地测试后，提交代码：

```
git commit -am "< 注释 >"
```

最后提交到 Gerrit 系统进行评审：

```
git review
```

提交到 Gerrit 后其他人员会进行评审，如图 10-22 所示。

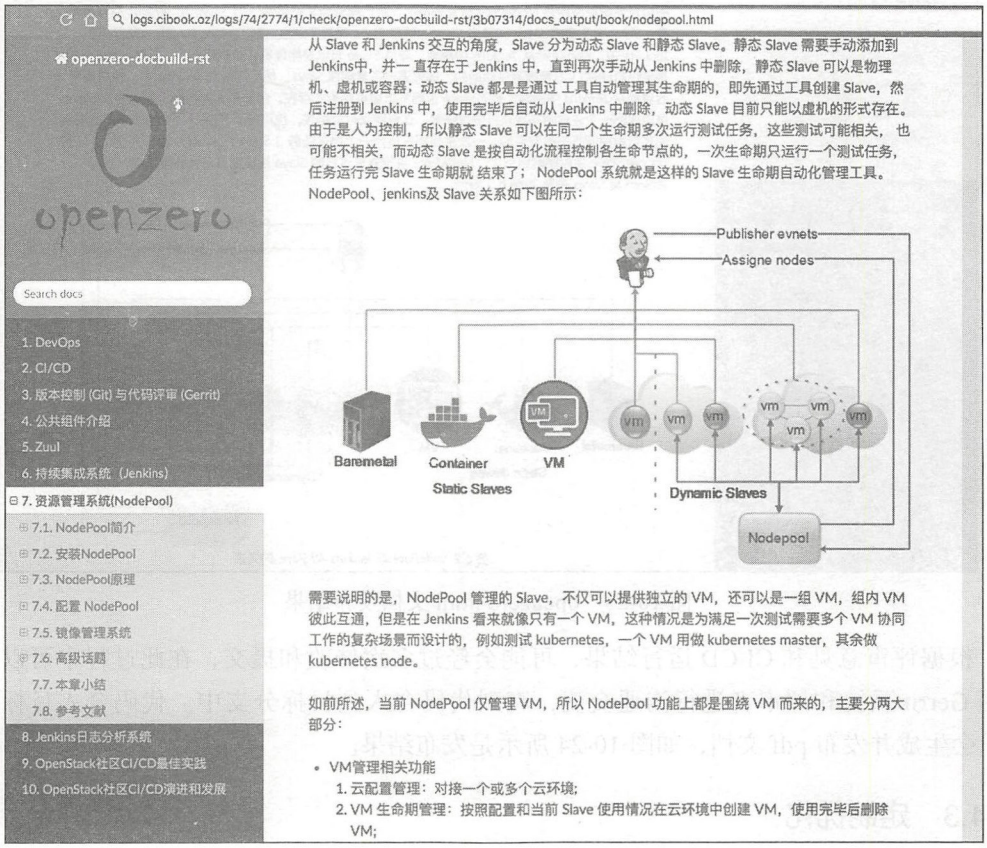


图 10-22　提交变更

openzero 项目每次变更提交后，都会发布临时 html 文档，通过 Job 运行结果中的链接可以看到 html 编译结果，如图 10-23 所示。

仅供非商业用途或交流学习使用

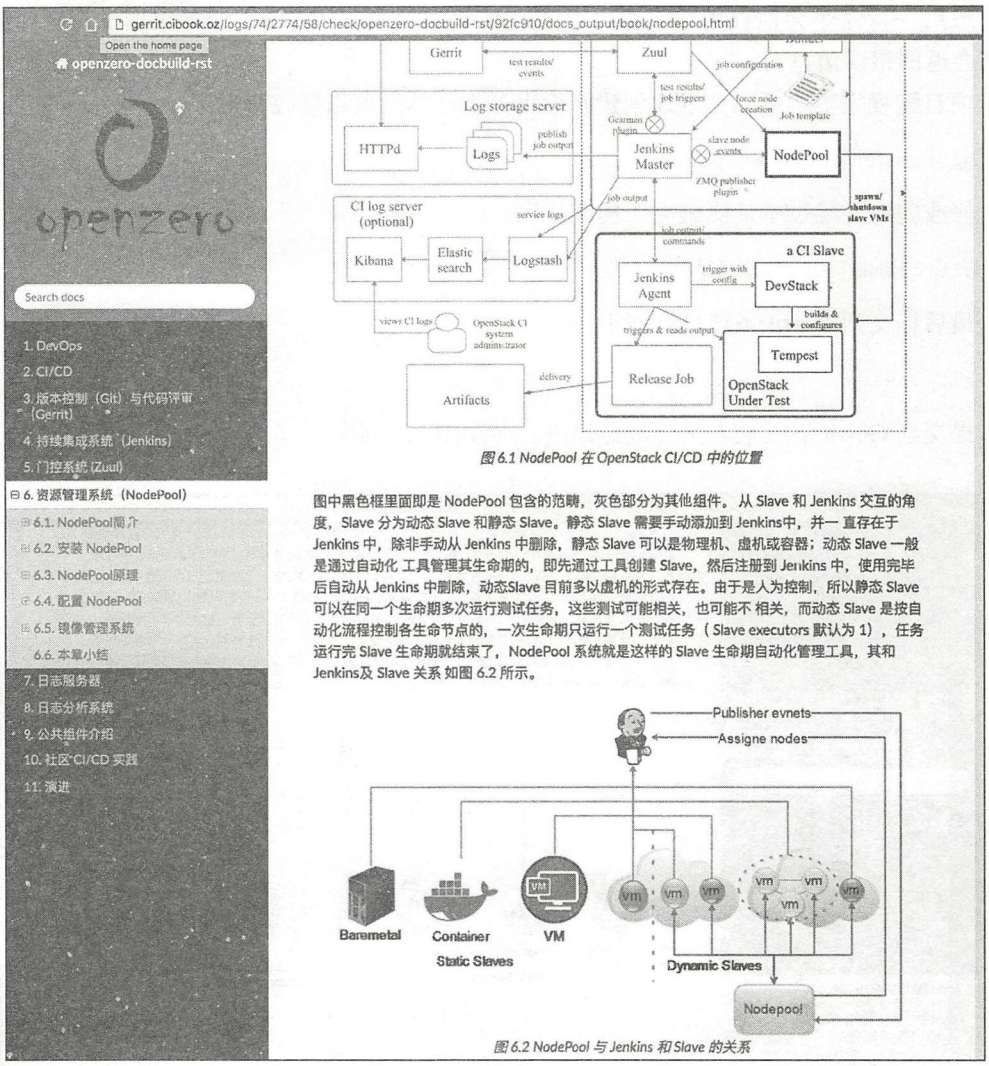


图 10-23　openzero html 文档发布结果

根据评审意见和 CI/CD 运行结果，可能会经过多次修改和提交，在此过程中可以充分利用 Gerrit 系统和评审者进行沟通交流，直到代码合入到目标分支中。代码合入目标分支后，会生成并发布 pdf 文档，如图 10-24 所示是发布结果。

10.4.3　定制优化

1. Puppet 模块定制

部署 CI master 时，需要从 Internet 上下载安装很多 Puppet 模块，导致安装时间非常长，而这些模块大部分是用来部署 OpenStack 基础设施或者 OpenStack 组件服务的，对于企业内部 CI 应用来说大部分不需要，因此可以裁减掉这些冗余的 Puppet 模块。

仅供非商业用途或交流学习使用

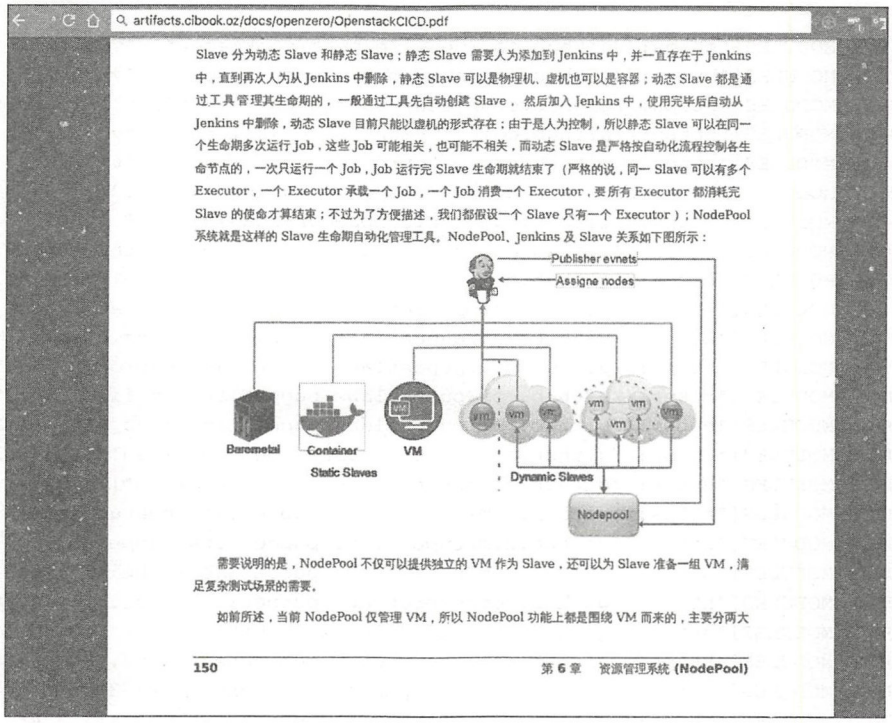


图 10-24　openzero pdf 文档发布结果

安装 Puppet 模块的配置文件位于 system-config/modules.env。裁剪之前的定制模块大致分为如下几部分：

- OpenStack 组件部署模块：包括 puppet-keystone、puppet-cinder、puppet-glance、puppet-ironic、puppet-neutron、puppet-nova、puppet-openstack_extras 和 puppet-openstacklib，如果企业内部不做 OpenStack 部署测试，这一部分都可删除。
- OpenStack-infra 组件部署模块：按照部署 CI 系统需要用到的组件服务裁剪，比如多节点系统部署需求，保留 puppet-gerrit、puppet-zuul、puppet-nodepool、puppet-jenkins 和 puppet-openstackci 等模块，其他可以删除。
- GitHub 上一些公共组件部署模块：一些公共组件部署模块，比如 Zookeeper、mysql、ntp 等，这些模块不在 OpenStack 的 git 库下。保留部署 CI 系统用到的公共组件模块即可，其他可以删除。

最终裁剪后的 Puppet 模块配置如代码清单 10-35 所示。

代码清单 10-35　裁剪后的 Puppet 模块配置

```
OPENSTACK_GIT_ROOT=https://git.openstack.org

# InfraCloud modules
SOURCE_MODULES["https://git.openstack.org/openstack-infra/puppet-vcsrepo"]=
```

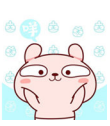
仅供非商业用途或交流学习使用

```
"0.0.8"
    SOURCE_MODULES["https://github.com/deric/puppet-zookeeper"]="v0.5.5"
    SOURCE_MODULES["https://github.com/duritong/puppet-sysctl"]="v0.0.11"
    SOURCE_MODULES["https://github.com/jethrocarr/puppet-initfact"]="1.0.1"
    SOURCE_MODULES["https://github.com/jfryman/puppet-selinux"]="v0.2.5"
    SOURCE_MODULES["https://github.com/maestrodev/puppet-wget"]="v1.6.0"
    SOURCE_MODULES["https://github.com/n1tr0g/golja-gnupg"]="1.2.2"
    SOURCE_MODULES["https://github.com/nanliu/puppet-staging"]="1.0.0"
    SOURCE_MODULES["https://github.com/puppetlabs/puppetlabs-apache"]="1.8.1"
    SOURCE_MODULES["https://github.com/puppetlabs/puppetlabs-apt"]="2.1.0"
    SOURCE_MODULES["https://github.com/puppetlabs/puppetlabs-concat"]="1.2.5"
    SOURCE_MODULES["https://github.com/puppetlabs/puppetlabs-firewall"]="1.1.3"
    SOURCE_MODULES["https://github.com/puppetlabs/puppetlabs-haproxy"]="1.5.0"
    SOURCE_MODULES["https://github.com/puppetlabs/puppetlabs-inifile"]="1.1.3"
    SOURCE_MODULES["https://github.com/puppetlabs/puppetlabs-java_ks"]="1.3.1"
    SOURCE_MODULES["https://github.com/puppetlabs/puppetlabs-mysql"]="3.6.2"
    SOURCE_MODULES["https://github.com/puppetlabs/puppetlabs-ntp"]="3.2.1"
    SOURCE_MODULES["https://github.com/puppetlabs/puppetlabs-postgresql"]="3.4.2"
    SOURCE_MODULES["https://github.com/puppetlabs/puppetlabs-puppetdb"]="3.0.1"
    SOURCE_MODULES["https://github.com/puppetlabs/puppetlabs-rabbitmq"]="5.2.3"
    SOURCE_MODULES["https://github.com/puppetlabs/puppetlabs-stdlib"]="4.10.0"
    SOURCE_MODULES["https://github.com/rafaelfelix/puppet-pear"]="1.0.3"
    SOURCE_MODULES["https://github.com/saz/puppet-memcached"]="v2.6.0"
    SOURCE_MODULES["https://github.com/saz/puppet-timezone"]="v3.3.0"
    SOURCE_MODULES["https://github.com/stankevich/puppet-python"]="1.9.4"
    SOURCE_MODULES["https://github.com/voxpupuli/puppet-alternatives"]="0.3.0"
    SOURCE_MODULES["https://github.com/voxpupuli/puppet-archive"]="v0.5.1"
    SOURCE_MODULES["https://github.com/voxpupuli/puppet-git_resource"]="0.3.0"
    SOURCE_MODULES["https://github.com/voxpupuli/puppet-nodejs"]="1.2.0"
    SOURCE_MODULES["https://github.com/voxpupuli/puppet-puppetboard"]="2.4.0"

    # Add modules that should be part of the openstack-infra integration test here
    # Please keep sorted
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-ansible"]="origin/
master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-cgit"]="origin/
master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-diskimage_builder"]=
"origin/master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-elastic_recheck"]=
"origin/master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-elasticsearch"]=
"origin/master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-ethercalc"]=
"origin/!master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-etherpad_lite"]=
"origin/master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-gerrit"]="origin/
master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-gerritbot"]=
"origin/master"
```

仅供非商业用途或交流学习使用

```
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-github"]="origin/
master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-httpd"]="origin/
master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-infracloud"]="origin/
master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-iptables"]="origin/
master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-jeepyb"]="origin/
master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-jenkins"]="origin/
master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-kerberos"]="origin/
master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-kibana"]="origin/
master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-log_processor"]=
"origin/master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-logrotate"]="origin/
master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-logstash"]="origin/
master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-mysql_backup"]=
"origin/master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-nodepool"]="origin/
master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-openstackci"]=
"origin/master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-pip"]="origin/master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-project_config"]=
"origin/master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-simpleproxy"]="origin/
master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-snmpd"]="origin/
master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-ssh"]="origin/master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-ssl_cert_check"]=
"origin/master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-sudoers"]="origin/
master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-user"]="origin/
master"
    INTEGRATION_MODULES["$OPENSTACK_GIT_ROOT/openstack-infra/puppet-zuul"]="origin/
master"
```

代码清单 10-35 所示的只是当前作者环境定制的情况，读者可以结合实际需要在社区默认配置的基础上增加或删除相应模块；此外，由于社区使用 OpenStack 主分支的模块配置，因此随着版本的不同，社区默认模块列表也可能会有变化。



仅供非商业用途或交流学习使用

2. 镜像裁剪和定制

Nodepool 编译的镜像是为 OpenStack 社区定制的，编译过程中在线下载安装了很多 OpenStack 项目的软件包以及相关的测试项目，存在以下问题：

- 镜像非常大，原始镜像 2G 左右；
- 编译时间非常长，差不多需要 4 小时，而且如果网络带宽不好的话，编译过程中任意一步失败，编译过程又需要重新开始；
- 在云环境上 launch 节点的时候，要执行定制化的操作，导致从节点 launch 到注册到 Jenkins 服务器的时间最长可超过 10 分钟；
- 镜像不满足测试场景的需求，缺少所需的软件包，比如 docker；当然可以在任务中按需安装，但不推荐使用这种做法。

针对以上问题，在企业内部应用中需要对镜像做裁剪和定制，下面分别对裁剪和定制做说明。

（1）裁剪

原有 Nodepool 配置的所有镜像 elements 如代码清单 10-36 所示。

代码清单 10-36　elements 列表

```
diskimages:
    - name: ubuntu-trusty
      elements:
          - ubuntu-minimal
          - vm
          - infra-package-needs
          - jenkins-slave
          - simple-init
          - cache-devstack
          - openstack-repos
          - nodepool-base
          - prepare-node
          - zuul-worker
          - growroot
          - stackviz
```

其中：

ubuntu-minimal：基础操作系统。

vm：设置分区磁盘。

simple-init：设置基础网络和系统配置。

growroot：扩展根分区。

以上几个是基础镜像 elements，不做裁剪。

- infra-package-needs：安装基础包，比如 git、cron、curl、git-review、tox、设置系统日志配置文件等，可以不修改。

- jenkins-slave：设置 Jenkins 用户下 git 的配置及用户登录的认证公钥等。
- openstack-repos：下载所有需要的 OpenStack 包，从 project-config/gerrit/projects.yaml 文件中读取所有的项目列表然后下载安装。社区默认配置有 2000 多个项目，大部分在企业 CI 中都用不到，删除所有不使用的项目，注册所需的项目即可。这一步可以大大减少镜像大小，缩短镜像编译时间。
- cache-devstack：缓存所有 devstack 可能需要用到的包和库，如果不做 OpenStack 部署，可以删除该 element。
- nodepool-base：处理镜像元数据以及一些和云相关的调整。
- prepare-node：准备节点，主要在 Jenkins 用户下创建了一些缓存文件夹，可以删除。
- zuul-worker：在 Zuul 用户下设置相关配置使 zuul-worker 处理程序可以远程登录，可以删除。
- stackviz：安装 stackviz 并构建 npm 依赖。

nodepool.yaml 的配置 label 下有一个 ready-script 参数，该参数配置为一个脚本名称，在节点孵化完成后注册到 Jenkins 之前调用该脚本执行，社区默认配置为 configure_mirror.sh。该脚本用来设置 DNS、检查 git.openstack.org 域名地址是否能正常访问、设置 pip 配置文件以及在系统的包管理工具中增加额外的包地址等。裁剪时删除大部分配置，修改 DNS 为企业内部域名服务器地址，如代码清单 10-37 所示。

代码清单 10-37　configure_mirror.sh

```
# set DNS address for openstack env
echo "nameserver 172.10.0.1" |sudo tee /etc/resolv.conf

echo "127.0.0.1 localhost"|sudo tee /etc/hosts
echo "127.0.1.1 ubuntu"|sudo tee -a /etc/hosts

cat << EOF > ~/.ssh/config
Host *
        StrictHostKeyChecking no
EOF
```

裁剪前后性能对比如表 10-2 所示，可以看出指标得到明显提升。

表 10-2　镜像裁剪前后性能对比

指标	裁剪前	裁剪后
镜像大小 (M)	1400	400
编译时间 (min)	240	40
注册时间 (min)	10	3

（2）定制

以安装 docker 为例，说明如何定制 elements。

因为已有名字为 docker 的 element，但是完成的不是安装 docker 的功能，所以这里不使用 docker 作为名字。新增名为 openzero 的 element，目录结构如代码清单 10-38 所示。

代码清单 10-38　openzero element

```
.
├── element-deps
├── install.d
│   └── 35-docker
├── README.rst
└── static
    └── etc
        └── docker
            └── daemon.json
```

因为是安装 docker 软件，所以对应 DIB 镜像制作 8 个阶段中的软件安装阶段，即需要增加 install.d 目录，该目录下存放执行的脚本代码，如代码清单 10-39 所示。

代码清单 10-39　安装 docker

```
#!/bin/bash

if [ ${DIB_DEBUG_TRACE:-0} -gt 0 ]; then
set -x
fi
set -eu
set -o pipefail
export SUDO='true'
export THIN='true'
sudo apt-get install -y apt-transport-https ca-certificates
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 \
    --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
echo "deb [arch=amd64] http://mirrors.oz/docker-ce xenial stable" | \
    sudo tee /etc/apt/sources.list.d/docker.list
sudo apt-get update
sudo apt-cache policy docker-engine
sudo apt-get update
sudo apt-get install -y linux-image-extra-$(uname -r) linux-image-extra-virtual
sudo apt-get install -y docker-engine
# Add jenkins user to docker group
sudo usermod -aG docker jenkins
```

接下来需要设置 docker 的配置文件，准备好配置文件 daemon.json，如代码清单 10-40 所示。其中 registry-mirrors 可设置为 docker 的官方镜像源，也可使用内部源。

代码清单 10-40　docker 配置文件

```
{
    "log-opts": {
        "max-size": "50m"
```

```
    },
    "registry-mirrors": [
        "https://dockerhub.oz"
    ],
    "log-driver": "json-file"
}
```

使用已有的 element install-static，该 element 的功能是复制所有 element 下 /static 目录的内容到镜像中。因此把准备好的配置文件 daemon.json 放在 openzero/static/etc/docker/ 目录下，镜像编译过程中会把 /etc/docker/daemon.json 原样复制到镜像中。因为 openzero 的 element 依赖于 install-static，还需要把 install-static 写入文件 element-deps，指明依赖关系。

针对以上镜像裁剪和定制的优化，最终在 nodepool.yaml 配置中指定的 element，如代码清单 10-41 所示，可以看到删除了 cache-devstack 和 prepare-node 两个 elements，新增了 openzero。

代码清单 10-41　定制后的 element 配置

```
diskimages:
    - name: ubuntu-trusty
      elements:
          - ubuntu-minimal
          - vm
          - infra-package-needs
          - jenkins-slave
          - simple-init
          - openstack-repos
          - nodepool-base
          - zuul-worker
          - growroot
          - stackviz
          - openzero
```

3. 使用内部 repo

要成功使用 OpenStack CI/CD，对网络有严格的要求，主要来自以下几个方面：

- Nodepool 在制作镜像过程中，需要从官网下载初始镜像和大量的安装包。
- Job 执行过程中由于任务的不同需要下载各种各样的安装包。
- 有些 Job 需要运行 docker，需要下载容器镜像。

从官网上下载大量的包，由于网络带宽限制会消耗一定的时间，而且如果网络不稳定会导致镜像制作或 Job 执行失败。解决这些问题的方式就是使用企业内部源，包括：

- apt/yum 源：软件包本地镜像，并保持和官网源同步。
- pip 源：python 软件包。
- docker 镜像源：容器镜像。

另外，对于常用的尤其是较大的文件，如果不属于以上类别，可以自己构建独立的 http server，作为文件服务器，以加速 CI/CD 运行过程中的访问。

如何搭建各 repo 源不在本文的介绍范围内，这里只介绍如何配置这些本地源。首先介绍下如何修改镜像制作的源。

（1）镜像制作源

在 Nodepool 的配置文件 nodepool.yaml 中，配置 diskimages 环境变量参数 DIB_DISTRIBUTION_MIRROR 为内部源地址，如代码清单 10-42 所示。

代码清单 10-42　镜像源定制

```
diskimages:
    - name: ubuntu-trusty
      rebuild-age: 864000
      elements:
          ...
      release: trusty
      env-vars:
          ...
          DIB_DISTRIBUTION_MIRROR: 'http://mirrors.cibook.oz/ubuntu'
          ...
```

示例中指定镜像编译使用的内部源为 http://mirrors.cibook.oz/ubuntu。

（2）虚机内部源

接下来介绍如何修改虚机内部使用的源，通常有两种方式：一种是修改 elements；另一种是在虚机启动之后注册到 Jenkins 之前调用脚本配置。

1）定制 element

element jenkins-slave 用来设置 Jenkins 用户下 git 的配置及用户登录的认证公钥等，可以在这里添加 / 修改各源的配置。

element jenkins-slave 的目录结构如代码清单 10-43 所示。

代码清单 10-43　jenkins-slave element

```
.
├── extra-data.d
│   └── 20-jenkins-user
└── install.d
    └── 20-jenkins-slave
```

以修改 pip 源为例，在 20-jenkins-slave 文件中添加如代码清单 10-44 所示的代码段。

代码清单 10-44　定制 pip 源

```
mkdir /home/jenkins/.pip
chmod 700 /home/jenkins/.pip

cat << EOF > /home/jenkins/.pip/pip.conf
[global]
index = http://nexus.oz/repository/pypi/
```

```
index-url = http://nexus.oz/repository/pypi/simple

[install]
trusted-host = nexus.oz
EOF
```

应用该 element 编译镜像时会在 Jenkins 用户下生成 pip 的配置文件，即修改了 pip 的源地址。其他修改如 apt/yum 源和 docker 源修改与此相同，不再赘述。

2）脚本配置

社区提供的 configure_mirror.sh 脚本支持通过注入环境变量的方式修改虚机的配置。即在 Nodepool 服务的节点上配置以 NODEPOOL_ 开始命名的环境变量，启动虚机后，Nodepool 会将这些环境变量注入虚机中，然后再执行 configure_mirror.sh，这样就可以定制虚机在运行过程中使用的 apt 源、pip 源和 docker 镜像源。其主要参数设置如代码清单 10-45 所示。

代码清单 10-45　脚本定制内部源

```
PIP_CONF="\
[global]
timeout = 60
index-url = $NODEPOOL_PYPI_MIRROR
trusted-host = $NODEPOOL_MIRROR_HOST
extra-index-url = $NODEPOOL_WHEEL_MIRROR"

PYDISTUTILS_CFG="\
[easy_install]
index_url = $NODEPOOL_PYPI_MIRROR
allow_hosts = *.openstack.org"

UBUNTU_SOURCES_LIST="\
deb $NODEPOOL_UBUNTU_MIRROR $LSBDISTCODENAME main universe
deb $NODEPOOL_UBUNTU_MIRROR $LSBDISTCODENAME-updates main universe
deb $NODEPOOL_UBUNTU_MIRROR $LSBDISTCODENAME-backports main universe
deb $NODEPOOL_UBUNTU_MIRROR $LSBDISTCODENAME-security main universe"

...

export NODEPOOL_DOCKER_REGISTRY_PROXY=${NODEPOOL_DOCKER_REGISTRY_PROXY: \
    -http://$NODEPOOL_MIRROR_HOST:8081/registry-1.docker}
```

如示例所示，只要在 Nodepool 节点上配置 NODEPOOL_PYPI_MIRROR、NODEPOOL_MIRROR_HOST、NODEPOOL_WHEEL_MIRROR 和 NODE-POOL_UBUNTU_MIRRO 等环境变量即可。

4. 多云接入

单机部署实践时 Nodepool 接入的是单个云环境，当单云故障或者云环境维护升级，当

前的 CI 系统就没有可用的 Slave 节点，所以支持多云环境接入变得很有必要，其系统拓扑结构如图 10-25 所示。

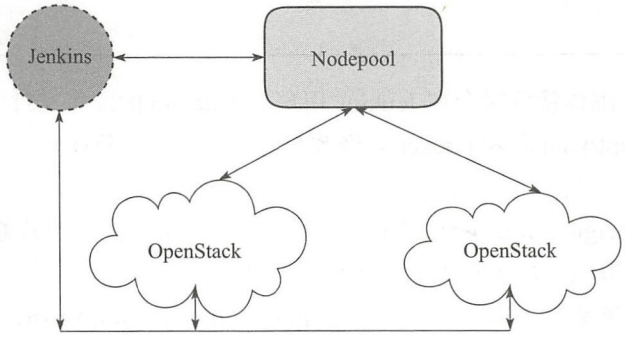
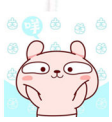


图 10-25　多云接入

在图 10-25 中，Nodepool 连接了两个云环境，当一个云环境出现异常时，Nodepool 可以使用另一个云环境为 Jenkins 服务。其配置如代码清单 10-46 所示，两个云环境分别为 zte-pod11-RegionOne 和 zte-pod13-RegionOne，总共提供 10 个带 ubuntu-trusty 标签的虚机和 6 个带 ubuntu-trusty-large标签的虚机，两个云环境都连接到同一个 Jenkins 服务器上。

代码清单 10-46　多云 Nodepool 配置

```
labels:
    - name: ubuntu-trusty
        image: ubuntu-trusty
        ready-script: configure_mirror.sh
        min-ready: 10
        providers:
            - name: zte-pod13-RegionOne
            - name: zte-pod11-RegionOne
    - name: ubuntu-trusty-large
        image: ubuntu-trusty-large
        ready-script: configure_mirror.sh
        min-ready: 6
        providers:
            - name: zte-pod13-RegionOne
            - name: zte-pod11-RegionOne
providers:
    - name: zte-pod13-RegionOne
        region-name: 'RegionOne'
        cloud: zte-pod13
        ...
        images:
            - name: ubuntu-trusty
              min-ram: 2048
              diskimage: ubuntu-trusty
              ...
            - name: ubuntu-trusty-large
```

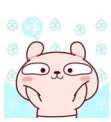


```
                min-ram: 8192
                diskimage: ubuntu-trusty
                ...
    - name: zte-pod11-RegionOne
        region-name: 'RegionOne'
        cloud: zte-pod11
        ...
        images:
            - name: ubuntu-trusty
                min-ram: 2048
                diskimage: ubuntu-trusty
                ...
            - name: ubuntu-trusty-large
                min-ram: 8192
                diskimage: ubuntu-trusty-large
                ...
targets:
    - name: jenkins1
        assign-via-gearman: True
        hostname: jenkins1-{label.name}-{provider.name}-{node_id}
```

代码清单 10-47 是代码清单 10-46 所示的两个云环境对应的云连接参数配置。

代码清单 10-47　多云环境连接配置

```
clouds:
    zte-pod11:
        profile: internap
        floating_ip_source: 'neutron'
        auth:
            username: 'ciuser'
            password: 'ciuser'
            project_name: 'ciuser'
            auth_url: 'http://openstack-11.cibook.oz:5000/v2.0'
        regions:
            - name: RegionOne
                values:
                    networks:
                    - name: admin_floating_net
                        routes_externally: true
                    - name: net-ci
                        nat_destination: true
                        default_interface: true
    zte-pod13:
        profile: internap
        floating_ip_source: 'neutron'
        auth:
            username: 'ciuser'
            password: 'ciuser'
            project_name: 'ciuser'
            auth_url: 'http://openstack-13.cibook.oz:5000/v2.0'
```



```
regions:
    - name: RegionOne
      values:
          networks:
    - name: admin_floating_net
      routes_externally: true
    - name: net-ci
      nat_destination: true
      default_interface: true
```

以上各项配置的具体含义参见第6章中的描述。需要说明的是对于多Nodepool服务的场景，为了避免各Nodepool管理的VM或镜像相互影响，建议在云环境中为不同的Nodepool配置不同的租户，这里给出的两个云环境中，用户和租户都是ciuser。连接两个云环境后，Jenkins上注册的部分VM如图10-26所示。

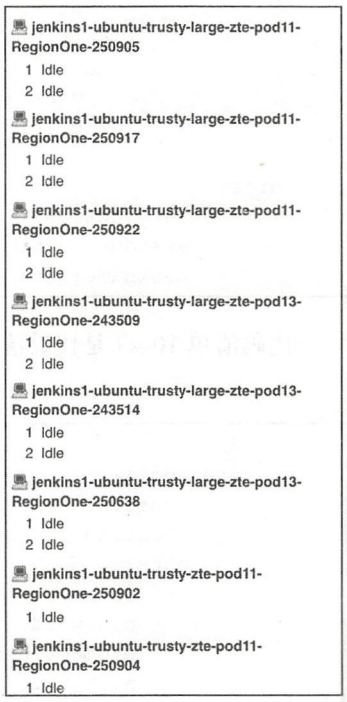


图10-26 多云接入

对于注册到Jenkins1上的VM，根据代码清单10-46中配置的VM命名规则hostname: jenkins1-{label.name}-{provider.name}-{node_id}，其名称中的pod11和pod13表明了VM所处的云环境为zte-pod11-RegionOne和zte-pod13-RegionOne。

5. 多Jenkins

在当前的OpenStack CI/CD中，Jenkins在整个系统中处于中心地位，它一定程度上决定着整个系统的性能。社区的测试结果显示，当Slave到达一定的数目（大约100左右）时，Jenkins的性能会急剧下降，进而成为整个系统的性能瓶颈。一个解决方式就是提高Jenkins所在服务器的性能，但这个提升有限，成本一般也较大，而且不符合云化的思想；另一个就是采用多Jenkins的方式，借助于Gearman的任务分发机制，同时提供多个Jenkins服务，每个Jenkins关于Job的配置相同，如图10-27所示。

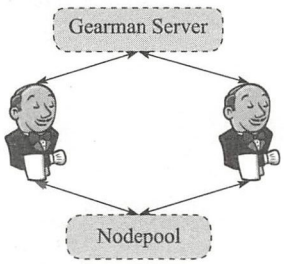


图10-27 多Jenkins

多Jenkins涉及的配置有以下几个方面：

- 首先，增加一个Jenkins的节点如jenkins2.oz，需要在master site.pp文件中添加类似代码清单10-48所示的信息。

代码清单10-48 增加jenkins2节点的配置

```
node 'jenkins2.oz' {
```

非卖品，仅供非商业用途或交流学习使用


```
        $vhost_name = hiera('vhost_name_jenkins2', $::fqdn)

        class { '::openstackci::jenkins_node':
            vhost_name              => $vhost_name,
            project_config_repo     => hiera('project_config_repo'),
            serveradmin             => hiera('serveradmin', "webmaster@${vhost_
name}"),
            jenkins_version         => hiera('jenkins_version', 'present'),
            jenkins_vhost_name      => hiera('jenkins_vhost_name2', 'jenkins2'),
            jenkins_username        => hiera('jenkins_username2', 'jenkins'),
            jenkins_password        => hiera('jenkins_password2', 'XXX'),
            jenkins_ssh_private_key => hiera('jenkins_ssh_private_key'),
            jenkins_ssh_public_key  => hiera('jenkins_ssh_public_key'),
            log_server              => hiera('log_server'),
            jjb_git_revision => hiera('jjb_git_revision', '1.6.2'),
            jjb_git_url             => hiera('jjb_git_url','https://git.openstack.
org/openstack-infra/jenkins-job-builder'),
            java_args_override      => hiera('java_args_override', '-Dhudson.model.,
ParametersAction.keepUndefinedParameters=true -Dorg.apache.commons.jelly.tags.fmt.
timeZone=Asia/Shanghai')
        }
    }

    node 'nodepool.oz' {
        # If the fqdn is not resolvable, use its ip address
        $vhost_name = hiera('vhost_name_nodepool', $::fqdn)

        class { '::openstackci::nodepool_node':
            vhost_name              => $vhost_name,
            project_config_repo     => hiera('project_config_repo'),
            jenkins_username        => hiera('jenkins_username', 'jenkins1'),
            jenkins_username2       => hiera('jenkins_username', 'jenkins2'),
            jenkins_ssh_public_key  => hiera('jenkins_ssh_public_key'),
            jenkins_ssh_private_key => hiera('jenkins_ssh_private_key'),
            oscc_file_contents      => hiera('oscc_file_contents', ''),
            mysql_host              => hiera('mysql_host'),
            mysql_root_password     => hiera('mysql_root_password'),
            mysql_nodepool_password => hiera('mysql_nodepool_password'),
            nodepool_jenkins_target
                                    => hiera('nodepool_jenkins_target', 'jenkins1'),
            jenkins_api_key         => hiera('jenkins_api_key', 'XXX'),
            jenkins_api_key2        => hiera('jenkins_api_key', 'XXX'),
            jenkins_credentials_id  => hiera('jenkins_credentials_id', 'XXX'),
            jenkins_credentials_id2 => hiera('jenkins_credentials_id2', 'XXX'),
            jenkins_url             => hiera('jenkins_url', 'http://localhost:8080'),
            jenkins_url2            => hiera('jenkins_url2', 'http://localhost:8080'),
            nodepool_revision       => hiera('nodepool_revision', 'master'),
            nodepool_git_source_repo
                                    => hiera('nodepool_git_source_repo','http://opnfv.
zte.com.cn/gerrit/openstack/nodepool'),
```

非卖品，仅供非商业用途或交流学习使用


```
        }
    }
```

代码清单 10-48 描述的信息增加了一个节点 jenkins2.oz，并针对 jenkins2 增加了一套参数 jenkins_vhost_name2、jenkins_url2、jenkins_api_key2 和 jenkins_credentials_id2，其他参数可以复用既有的 Jenkins 的配置。同时需要修改已有 nodepool.oz 节点，使其能够连接 jenkins2.oz（openstackci::nodepool_node 需要做同样的修改）。

- 接着，在 master 的 common.yaml 中增加新 Jenkins2 的连接参数，如代码清单 10-49 所示。

代码清单 10-49　master 增加一套 Jenkins 参数配置

```
jenkins_vhost_name2: jenkins2.oz
jenkins_url2: http://jenkins2:8080
jenkins_api_key2: ef11cfaebab8659d7f8a3e7cb0649a6e
jenkins_credentials_id2: 0f779e44-33f4-4266-8964-d729ab16c15a
```

注解　jenkins_api_key2 和 jenkins_credentials_id2 的配置需要手动进行，具体参见 10.2.2 节中的描述。

- 最后，修改 nodepool.yaml 中的配置，代码清单 10-50 所示。

代码清单 10-50　多 Jenkins 的 Nodepool 配置

```
...
targets:
    - name: jenkins1
        assign-via-gearman: True
        hostname: jenkins1-{label.name}-{provider.name}-{node_id}
    - name: jenkins2
        assign-via-gearman: True
        hostname: jenkins2-{label.name}-{provider.name}-{node_id}
...
```

jenkins2 对应的连接参数，Puppet Agent 同步后会自动生成到 /etc/nodepool/secure.conf 文件中，如代码清单 10-51 所示。

代码清单 10-51　Nodepool 中 Jenkins 连接参数

```
...
[jenkins "jenkins1"]
user=jenkins
apikey=fd11cfaebab8659d7f8a3e7cb0649a6d
credentials=0e779e44-eb45-4266-864c-d729ab16c15a
url=http://jenkins.oz:8080
```

非卖品，仅供非商业用途或交流学习使用


```
[jenkins "jenkins2"]
user=jenkins
apikey=ef11cfaebab8659d7f8a3e7cb0649a6e
credentials=0f779e44-33f4-4266-8964-d729ab16c15a
url=http://jenkins2.oz:8080
```

6. 多 Gerrit

对于大多数企业来说，多 Gerrit 的需求比较常见，原因也各不相同：

1）异地开发，基于运维原因都在本地部署一个 Gerrit。

2）出于对安全的考量，将对内对外的 Gerrit 服务分离。

3）基于组织结构，不同的项目或产品都自己独立的 Gerrit 系统。

以上情况为了充分利用资源或者为集中统一管理，可能需要公用 CI/CD 的其他子系统，如公用 Jenkins 和 Nodepool。当前版本 Zuul 还不支持多 Gerrit 接入（Zuul V3 会引入这个新特性），所以对于多 Gerrit，当前的推荐方案是多 Zuul，即一个 Gerrit 对接一个 Zuul，如图 10-28 所示。

多 Gerrit 配置需要增加一个 Zuul 节点，如果 Gerrit 不是现有的也需要增加，两者的配置基本上是现有 Gerrit 和 Zuul 配置的复制并做简单修改，在此不再详细描述。

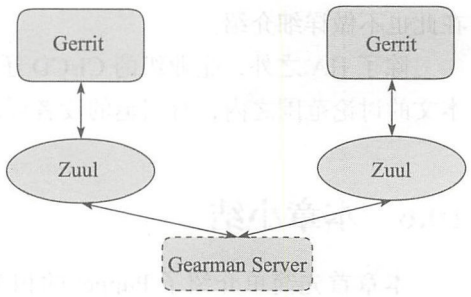


图 10-28　多 Gerrit

10.5　CI/CD 还需要考虑的问题

多节点部署不仅解决了 OpenStack CI/CD 组件间耦合的问题，更为整个 CI/CD 系统性能的提升以及水平扩展提供了可能。对于企业级的 CI/CD，高可用性（High Available，HA）也必不可少，否则一个组件服务宕机会导致整个 CI/CD 系统都没法工作。有了前文多节点方案以及扩展性和性能的讨论，基于 OpenStack CI/CD 的 HA 的方案也呼之欲出了，如图 10-29 所示是作者提出的 HA 方案，待实践验证。

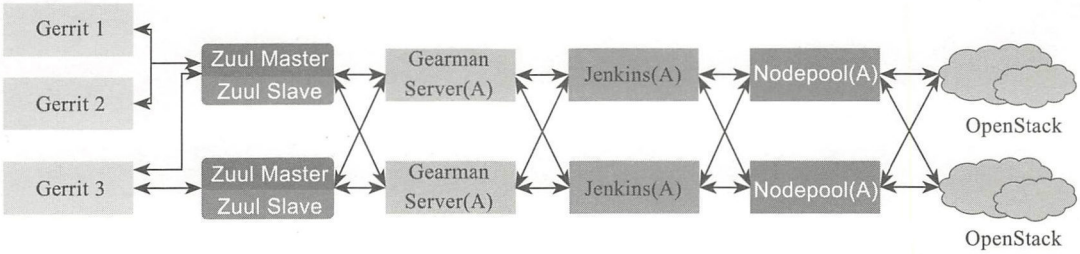


图 10-29　基于 OpenStack CI/CD 的 HA

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

由图 10-29 可知，Gearman、Jenkins 和 Nodepool 都使用了 Active-Active 方式，并且每类组件都分别提供了两个服务节点，这样某类组件如果其中一个节点因为异常不能正常工作，其服务可以由另一个节点接替。Zuul 相对比较特殊，只能使用 Active/Standby（也叫 Master/Slave）方式，即 Master 处于 Active 状态，Slave 处于 Standby 状态，Master 和 Slave 保持状态同步。当 Master 出现异常或升级时 Slave 会成为 Master，从而保持 Zuul 服务的连续性；对于连接不同 Gerrit 的 Zuul，分别提供 Master/Slave 对，不同 Zuul Master/Slave 对之间彼此独立。

图 10-29 中接入了两个 OpenStack 环境提供 Slave 节点，每个 OpenStack 同时接收两个 Node-pool 管理，为避免 VM 浪费，每个 OpenStack 提供一半的 Slave。

需要特别说明的是，对于 Gerrit，只能接入一个 Zuul，连接多个 Zuul 门控系统可能会带来运行结果的冲突。此外，ELK 的 HA 都可以独立形成集群，已有解决方案，故不在讨论范围之内；LogServer 也比较独立，可以根据需要部署多个，且每个 LogServer 独立配置，在此也不做详细介绍。

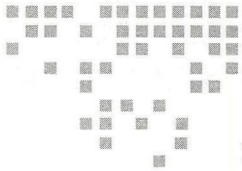
除了 HA 之外，企业级的 CI/CD 还需要对接企业的认证系统，尤其是 Gerrit，但这不在本文的讨论范围之内，有兴趣的读者可以参考 Gerrit 官方文档。

10.6 本章小结

本章首先简单介绍了 Puppet 的相关概念；然后依据社区的指导文档实践了 OpenStack CI/CD 的单机部署，对相关配置做了较为详细的讲解；接着为了满足生产环境部署的需要，给出了多节点的部署方案和相关配置指导，并站在项目管理者、普通开发者和基础设施维护者的角度，概要阐述了如何在 CI/CD 系统中增加新项目、贡献代码以及系统的维护；最后讨论了在使用该系统框架时企业还需要考虑的典型问题，以及推荐方案。通过本章的学习，希望读者能搭建一个满足企业内部需要的 CI/CD 环境。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用



第 11 章 Chapter 11

演　进

随着 OpenStack 社区项目持续增加、技术不断演进和发展，对基础设施也提出了更多的需求和挑战。为了使基础设施更简单、灵活，易于使用和扩展，CI/CD 技术也在不断往前演进和发展。

本章重点介绍 V2 版本 CI/CD 系统中存在的问题及针对这些问题社区做的技术演进和未来的发展方向。

11.1　存在的问题

首先，Zuul 的项目配置不分租户，所有信息共享不利于管理和维护，同时 Job 的配置仓库和项目仓库分离，不利于测试；其次，CI/CD 中实际使用到的 Jenkins 功能比较少，并且 JJB 需要适配 Jenkins 的版本演进；最后，Nodepool 对于支持多节点测试比较低效，而且当前只支持虚机方式的资源池已经不能满足部分场景测试需求。

11.1.1　耦合

Zuul V2 版本中，所有项目的 Job 配置在公共的 openstack-infra/project-config 仓库中，项目代码本身有自己的仓库，建立一个项目的 CI 需要分两步进行：

- 在公共仓库的 jenkins/jobs 中定义 Job，并在该库的 zuul/layout.yaml 文件中增加对应 pipeline 下该 Job 的引用。
- 项目中持续集成直到 Job 运行正常。

这种模式的缺点是，通常情况下开发人员在不同项目的权限不同，项目开发人员在公

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

共仓库中提交本项目的 Job 定义，需要公共项目库的核心开发人员批准才能合入，合入之后才能在项目库中调试，非常不便于项目人员修改和测试 Job；同时公共项目库的开发人员其实应该不关心项目具体 Job 的实现，这种模式增加了不同项目开发人员的工作量。

11.1.2 Zuul

Zuul V2 只支持对接一个 Gerrit，没有租户的概念，所有的项目配置都存储在同一个文件 layout.yaml 中。而随着需求的演进，支持对接多个 Gerrit 服务器或者其他代码管理系统很有必要。这种场景下如果所有的配置存储在一个文件中则不易于管理和维护。而且对于具有相似行为的一组项目可能并不关心其他不相关项目的配置信息，同时由于不同代码管理系统的触发条件不同，pipeline 的配置有可能冲突。因此在该应用场景下需要支持对项目分租户，在同一租户下共享相同的配置信息，租户可以自行定义需要管理哪些项目。比如所有 OpenStack 相关被管理的项目放在 openstack 租户下。

这里租户的概念和 OpenStack 云环境中的租户概念不一样，指共享 Zuul 配置的一组项目的集合。

11.1.3 Jenkins

OpenStack CI/CD 使用的 Jenkins 功能很少，JJB 的定义也比较复杂，其中一些复杂性来自需要支持特定的 Jenkins 行为，由于 Jenkins 的版本更新，JJB 可能需要重新适配。

11.1.4 Nodepool

1. 多节点测试低效

虽然 V2 版本已经做了扩展，采用特定标签的方式支持单个任务多节点测试，但过程非常低效：

- 需要同时创建该标签下的所有节点，可能会导致创建节点延迟；
- 不能随着使用而扩展，因为它们仅适用于该标签下的多节点任务，该标签下提供的节点数量在运行过程中不能增减，不能用于其他多节点任务，因此节点大部分时间可能处于空闲状态，造成资源浪费。比如有两个多节点任务 A 和 B，A 任务运行在 A（3 个节点）标签上，B 任务运行在 B（5 个节点）标签上，当前有 B 标签的资源池空闲，当需要构建 A 任务时，空闲的 5 个节点可以满足 3 个节点测试需求，但是由于 A 任务指定了 A 节点标签而无法调度到 B 节点标签上运行，这就造成了资源的浪费。

2. 单类型的动态 Slave 节点

随着测试需求的增多，有些测试场景需要运行在裸机或者容器上，Nodepool 单一的动态虚机节点管理已经不能满足测试需求，需要增加对裸机和容器的支持。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

11.2 演进

11.2.1 架构

针对上述存在的问题和需求，社区基础架构团队在 V2 版本上进行了大量变更和改进，移除了 Jenkins 组件，增强了 Nodepool 功能，并重写了 Zuul，即 Zuul V3。

系统架构如图 11-1 所示，可以看出整体框架改动不少。

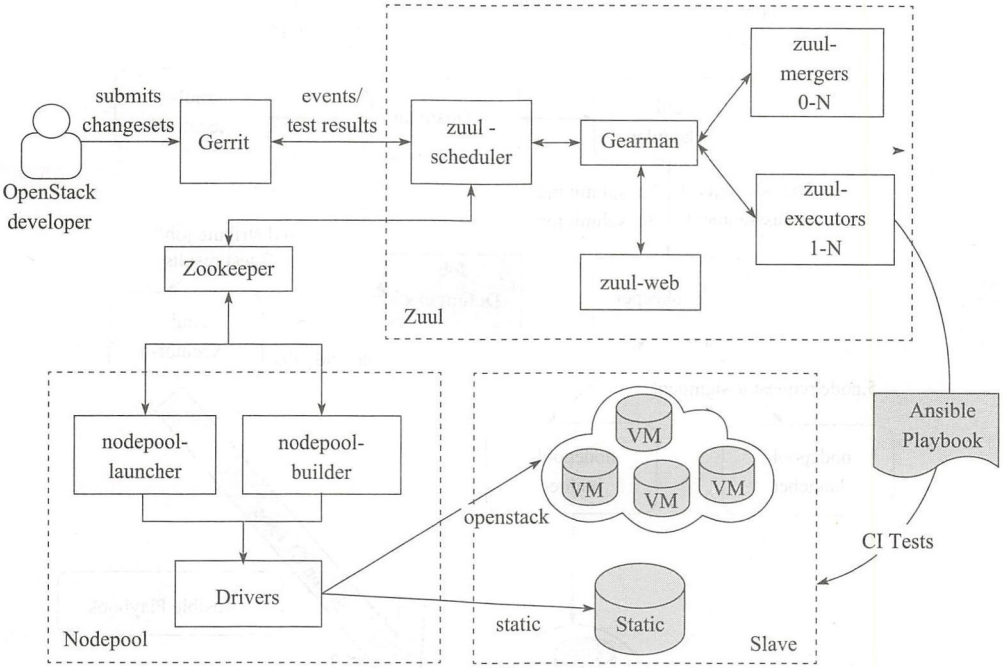


图 11-1 Zuul V3 系统架构

Zuul V3 有四个组件服务，内部通过 Gearman 通信，分别完成如下功能：

- zuul-scheduler：主处理服务，和 V2 版本的 zuul-server 服务功能大致相同，负责接收外部事件、驱动执行 Job、收集和上报结果，以及协调其他服务的工作。
- zuul-merger：执行一系列 git 操作，和 V2 没有区别；新增可弹性伸缩配置，可选 0 或者多个。
- zuul-executor：新增服务，执行 Job 的服务，和 zuul-merger 一样可弹性伸缩，至少配置 1 个服务，如果没有配置 zuul-merger，由 zuul-executor 完成 zuul-merger 的功能。
- zuul-web：新增 Web 服务，提供 RESTful API。

移除了 Jenkins，Zuul 通过 Zookeeper 与 Nodepool 交互请求节点信息。

Nodepool V3 有两个组件服务，内部通过 Zookeeper 通信，分别完成如下功能。

- nodepool-launcher：和 V2 版本的 Nodepool 服务相同，管理云虚机，区别在于虚机

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

信息由原来存储在数据库中改为存储在 Zookeeper 中。

❑ nodepool-builder：构建和上传镜像到云环境，和 V2 版本无差别。

整体工作流如图 11-2 所示，具体分析如下：

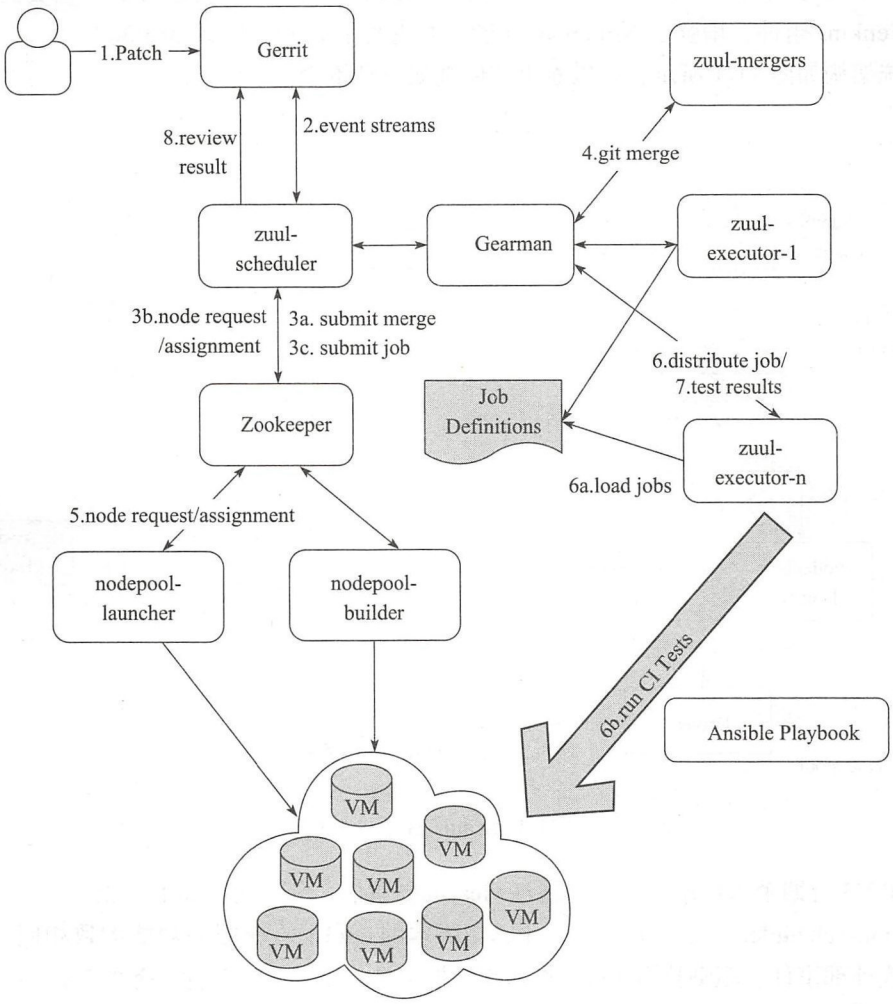


图 11-2　V3 工作流程

1）贡献者在代码管理系统上操作：比如在 Gerrit 上提交新 patch，添加评论或者合入 patch 等；

2）Gerrit 产生一个通知事件到它的事件流中（event stream）；

3）zuul-scheduler 从事件流中读取事件，匹配事件到一个或多个 pipeline，找到该 patch 对应的项目下的 pipeline 任务，依次完成如下操作：

❑ 通过 Gearman 向 zuul-merger 提交代码合并任务；

仅供非商业用途或交流学习使用

❑ 通过 Zookeeper 向 nodepool-launcher 请求任务节点；

❑ 通过 Gearman 向 zuul-exectutor 提交执行 Job。

4）zuul-merger 完成代码合并；

5）nodepool-launcher 根据节点请求信息分配相应的节点并返回节点信息；

6）zuul-executor 根据任务信息动态加载任务定义，并生成 Ansible Playbook，使用 Ansible 在对应的节点上运行测试；

7）zuul-executor 通过 Gearman 返回测试结果到 zuul-scheduler；

8）根据测试结果，zuul-scheduler 在 Gerrit 中对 patch 添加一个 review 结果。

整体工作流和 V2 版本的主要差别在于任务的加载、节点的请求和分配以及任务的执行方式。下面介绍下每个组件支持的新特性。

11.2.2 Zuul V3

1. 分布式配置

Zuul V3 版本中，项目的 Job 配置分布在各个项目的仓库中，在变更合并前测试，简化了测试工作流程。openstack-infra/zuul-jobs 是默认 Job 配置库，包含常用 Job 定义，Zuul 仅需导入本地配置中尚未定义的 Job。这对于第三方 CI 来说是个好消息，因为 V2 版本中需要在第三方 CI 系统的 Jenkins 上配置所有 Job，而新框架下只需配置第三方 Job，重用上游的公共 Job，如 tox-docs 或 tox-py35 等。

2. 多租户配置

项目按租户配置划分，同一个租户下共享相同的配置信息。

3. 多代码管理系统的集成

V2 版本只支持和单个 Gerrit 集成，新版本不仅支持多 Gerrit 对接，而且集成多代码管理系统，比如 github，更加灵活可扩展。

4. Ansible Job 定义

使用 Ansible 作为运行 CI 测试的基础，Job 在 Ansible 中创建具有如下优势：

❑ 多节点架构，Job 易于分发；

❑ Ansible 模块生态系统简化了复杂的 Job；

❑ Playbook 可以手动执行；

❑ 更容易获得构建日志。

5. Zuul 新增服务

新增 zuul-executor 服务执行原来由 Jenkins 完成的 Job 加载和执行的功能；新增 zuul-web 服务提供 RESTful API，包括租户信息和状态查询，指定租户下的变更、Job 和构建查询等。

仅供非商业用途或交流学习使用

11.2.3 Nodepool V3

1. 显式节点操作

支持显式的节点创建和删除。对于多节点测试，此模型效率更高，不再需要特殊的多节点标签。按照不同 Job 配置来请求可用节点资源池。如果当前可用的节点配置和数目满足要求，则分配节点做测试；不满足则按照配置信息创建新的所需节点。这种模型下节点的需求和分配计算更简单。

2. 支持更多类型的资源池

原来支持基于 OpenStack 创建的动态节点资源池不变，增加一个基于驱动程序的设计来支持非 OpenStack 的 Provider，该接口通用，可用于实现新的 Provider，比如容器或静态节点等，目前已经支持静态节点的驱动[⊖]。

11.3 CI/CD 发展

V3 版本虽然在 V2 上做了很多改进，但是在大规模的迁移和使用过程中，仍然暴露出一些问题和需求。

1. 无服务中断

当前如果用户修改 Zuul 的配置，需要重启服务才能生效，如果用户频繁更新配置，必然引起服务中断。无服务中断是 Zuul 下一个版本主要解决的问题，通过扩展 Zuul 的 scheduler 服务来避免宕机和频繁更新。

2. 任务在指定租户下运行

对于多租户的云环境，特定租户下的节点类型只能由特定租户使用，对于这种应用场景，如何在 Job 中指定租户下的节点，也是社区当前热门的讨论话题。

3. 支持容器 /Baremetal 作为 Slave 节点

随着容器技术的成熟，Nodepool 支持容器的 slave 节点也是大势所趋，这一功能正在开发中。而 Baremetal 的支持由于应用场景比较复杂，比如静态节点也可以是 Baremetal，或者通过 openstack + ironic、ironic + bifrost、cabbler、maas 等驱动方式，社区正在考虑先文档化这些驱动方式配置，再由用户按照需求实现对应的驱动即可。

11.4 本章小结

本章主要介绍了 Zuul V3 版本的新特性以及未来的技术演进方向（即 Zuul V4），感兴趣的读者可以直接参与到社区中讨论和贡献。

⊖ https://review.openstack.org/#/c/535553/。

仅供非商业用途或交流学习使用

作者介绍

董文娟

中兴通讯高级软件工程师，2015年加入开源社区，从事故障管理和根因分析等相关工作，是OpenStack社区Vitrage项目和OPNFV社区Doctor项目的核心贡献者。一直从事企业内部CI/CD方面的研究工作，对CI/CD的基础设施搭建有丰富的实践经验，致力于研究构建企业级高性能、高可靠的DevOps平台的解决方案。

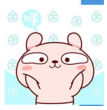
尚小冬

中兴通讯资深系统工程师，近10年通信系统产品平台研发经验。2015年加入OpenStack、OPNFV和ONAP等开源社区，目前为OPNFV社区Parser项目PTL。对OpenStack各个核心组件和OpenStack CI/CD均有深入理解和深度实践，目前聚焦于Cloud Native相关产品的设计和实现。

张　军

中兴通讯首席架构师，超过15年的通信系统的设计和研发经验，拥有十多项专利。2014年起，带领团队进行NFV转型，负责从基于OpenStack的IaaS平台到基于Kubernetes的PaaS平台的架构设计。带领团队积极参与OpenStack、OPNFV和CNCF等外部社区贡献，在公司内承担开源布道的工作，推动公司研发转型。目前是中兴通讯在OPNFV的TSC代表和Pharos项目PTL。

仅供非商业用途或交流学习使用



OpenStack 作为基于社区开发的大型开源项目的典范，在过去的七八年里获得了巨大的成功。在产品背后，支撑其稳步发展的是一套高度自动化的持续集成系统。这套系统能够让上千名贡献者协同工作，一起构建一个如此复杂的基础设施（即服务产品），说明它必有其过人之处。本书将带你洞悉其中的各个细节，对企业构建满足自身需求的持续集成系统具有很大的借鉴价值！

——**孙振鹏** 国际最佳实践管理联盟主席/EXIN 国际信息科学考试学会亚太区总经理

在敏捷开发大行其道的时代，CI/CD的自动化变得非常重要。本书深入介绍了OpenStack CI/CD系统的组成部分及相关实践，对于致力于构建DevOps系统的企业和软件开发人员来说，具有重要的借鉴意义！

——**何宝宏** 中国信息通信研究院云计算与大数据研究所所长

随着云计算的兴起，软件架构已经发生了巨大的变化，对集成测试提出了更大的挑战。本书选取了典型的云计算平台OpenStack 的CI/CD架构，由浅入深，由理论到实践，详细阐述了OpenStack CI/CD系统及各组件的功能，并应用该系统定制企业级CI/CD，对建设企业级DevOps有很好的借鉴意义。

——**袁春风** 南京大学计算机系教授

OpenStack是著名的开源云计算管理平台，CI/CD是其关键特性。本书的作者团队是一支卓越的敏捷自组织团队，他们深耕社区，回馈社会。他们创作的《OpenStack CI/CD：原理与实践》全面剖析了OpenStack CI/CD的组成部分，既有深入浅出的理论介绍，又有生动丰富的案例分享，是本领域的必备佳作。

——**闫林** 中兴通讯IT技术学院副院长

上架指导：计算机/程序设
ISBN 978-7-111-6119
9 787111 6119
定价：69.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn